

The Fortress Language Specification

Version 1.0 β

Eric Allen
David Chase
Joe Hallett
Victor Luchangco
Jan-Willem Maessen
Sukyoung Ryu
Guy L. Steele Jr.
Sam Tobin-Hochstadt

Additional contributors:

Joao Dias
Carl Eastlund
Christine Flood
Yossi Lev
Cheryl McCosh

© Sun Microsystems, Inc.

March 6, 2007

Contents

I Preliminaries	14
1 Introduction	15
1.1 Fortress in a Nutshell	15
1.2 Acknowledgments	17
1.3 Organization	17
2 Overview	18
2.1 The Fortress Programming Environment	18
2.2 Exports, Imports, and Linking Components	20
2.3 Automatic Generation of APIs	23
2.4 Rendering	23
2.5 Some Common Types in Fortress	24
2.6 Functions in Fortress	25
2.7 Some Common Expressions in Fortress	27
2.8 For Loops Are Parallel by Default	28
2.9 Atomic Expressions	28
2.10 Dimensions and Units	28
2.11 Aggregate Expressions	29
2.12 Comprehensions	30
2.13 Summations and Products	31
2.14 Tests and Properties	31
2.15 Objects and Traits	32
2.16 Features for Library Development	35
II Fortress for Application Programmers	37
3 Programs	38

4	Evaluation	39
4.1	Values	39
4.2	Normal and Abrupt Completion of Evaluation	40
4.3	Memory and Memory Operations	40
4.4	Threads and Parallelism	41
4.5	Environments	44
4.6	Input and Output Actions	44
5	Lexical Structure	45
5.1	Characters	45
5.2	Words and Chunks	48
5.3	Lines, Pages and Position	48
5.4	ASCII Conversion	48
5.5	Input Elements and Scanning	49
5.6	Comments	50
5.7	Whitespace Elements	50
5.8	Reserved Words	50
5.9	Character Literals	51
5.10	String Literals	52
5.11	Boolean Literals	53
5.12	The Void Literal	53
5.13	Numerals	53
5.14	Operator Tokens	54
5.15	Identifiers	55
5.16	Special Tokens	55
5.17	Rendering of Fortress Programs	55
6	Declarations	58
6.1	Kinds of Declarations	58
6.2	Top-Level Variable Declarations	60
6.3	Local Variable Declarations	61
6.4	Local Function Declarations	62
6.5	Matrix Unpasting	62

7	Names	66
7.1	Namespaces	66
7.2	Reach and Scope of a Declaration	66
7.3	Qualified Names	68
8	Types	69
8.1	Relationships between Types	69
8.2	Trait Types	70
8.3	Object Trait Types	70
8.4	Tuple Types	70
8.5	Arrow Types	71
8.6	Bottom Type	72
8.7	Types in the Fortress Standard Libraries	72
8.8	Intersection and Union Types	72
8.9	Type Aliases	73
9	Traits	74
9.1	Trait Declarations	74
9.2	Method Declarations	76
9.3	Abstract Field Declarations	79
9.4	Method Contracts	80
9.5	Value Traits	81
10	Objects	82
10.1	Object Declarations	82
10.2	Field Declarations	84
10.3	Value Objects	84
10.4	Object Equivalence	85
11	Static Parameters	86
11.1	Type Parameters	86
11.2	Nat and Int Parameters	87
11.3	Bool Parameters	87
11.4	Dimension and Unit Parameters	87
11.5	Operator and Identifier Parameters	88
11.6	Where Clauses	89

12 Functions	91
12.1 Function Declarations	91
12.2 Function Applications	93
12.3 Abstract Function Declarations	95
12.4 Function Contracts	95
13 Expressions	97
13.1 Literals	97
13.2 Identifier References	99
13.3 Dotted Field Accesses	99
13.4 Dotted Method Invocations	100
13.5 Naked Method Invocations	100
13.6 Function Calls	101
13.7 Function Expressions	101
13.8 Operator Applications	101
13.9 Object Expressions	102
13.10 Assignments	103
13.11 Do Expressions	104
13.12 Label and Exit	106
13.13 While Loops	107
13.14 Generators	107
13.15 For Loops	108
13.16 Ranges	108
13.17 Summations and Other Reduction Expressions	109
13.18 If Expressions	110
13.19 Case Expressions	111
13.20 Extremum Expressions	112
13.21 Typecase Expressions	112
13.22 Atomic Expressions	113
13.23 Spawn Expressions	114
13.24 Throw Expressions	115
13.25 Try Expressions	115
13.26 Static Expressions	116
13.27 Aggregate Expressions	118

13.28	Comprehensions	121
13.29	Type Ascription	122
13.30	Type Assumption	122
13.31	Expression-like Functions	122
14	Exceptions	125
14.1	Causes of Exceptions	125
14.2	Types of Exceptions	125
14.3	Information of Exceptions	126
15	Overloading and Multiple Dispatch	128
15.1	Terminology and Notation	128
15.2	Applicability to Named Functional Calls	129
15.3	Applicability to Dotted Method Calls	129
15.4	Applicability for Functionals with Varargs and Keyword Parameters	130
15.5	Overloading Resolution	130
16	Operators	132
16.1	Operator Names	132
16.2	Operator Precedence	133
16.3	Operator Fixity	135
16.4	Chained and Multifix Operators	136
16.5	Enclosing Operators	136
16.6	Conditional Operators	137
16.7	Juxtaposition	137
16.8	Overview of Operators in the Fortress Standard Libraries	139
17	Conversions and Coercions	144
17.1	Principles of Coercion	144
17.2	Coercion Declarations	145
17.3	Coercion Invocations	146
17.4	Applicability with Coercion	147
17.5	Coercion Resolution	148
17.6	Restrictions on Coercion Declarations	150
17.7	Coercions for Tuple and Arrow Types	150
17.8	Automatic Widening	151

18 Dimensions and Units	154
19 Tests and Properties	158
19.1 The Purpose of Tests and Properties	158
19.2 Test Declarations	158
19.3 Other Test Constructs	159
19.4 Running Tests	159
19.5 Test Suites	160
19.6 Property Declarations	160
20 Type Inference	162
20.1 What Is Inferred	162
20.2 Type Inference Procedure	162
20.3 Finding “Closest Expressible Types” for Inferred Types	164
21 Memory Model	165
21.1 Principles	165
21.2 Programming Discipline	166
21.3 Read and Write Atomicity	168
21.4 Ordering Dependencies among Operations	168
22 Components and APIs	171
22.1 Overview	171
22.2 Components	172
22.3 APIs	175
22.4 Tests in Components and APIs	176
22.5 Type Inference for Components	177
22.6 Initialization Order for Components	177
22.7 Basic Fortress Operations	177
22.8 Advanced Features of Fortress Operations	185
III Fortress APIs and Documentation for Application Programmers	189
23 Objects	190
23.1 The Trait <code>Fortress.Core.Any</code>	190
23.2 The Trait <code>Fortress.Core.Object</code>	191
23.3 The Trait <code>Fortress.Core.Tuple</code>	192

24	Booleans and Boolean Intervals	193
24.1	The Trait Fortress.Core.Boolean	193
24.2	The Trait Fortress.Standard.BooleanInterval	197
24.3	Top-level BooleanInterval Values	204
25	Numbers	205
25.1	Rational Numbers	205
25.2	Integers	213
26	Negated Relational Operators	225
26.1	Negated Equivalence Operators	225
26.2	Negated Comparison Operators	226
27	Exceptions	228
27.1	The Trait Fortress.Standard.Exception	228
27.2	The Trait Fortress.Standard.CheckedException	228
27.3	The Trait Fortress.Standard.UncheckedException	229
28	Threads	230
28.1	The Trait Fortress.Standard.Thread	230
29	Dimensions and Units	231
29.1	Fortress.SIUnits	231
29.2	Fortress.EnglishUnits	233
29.3	Fortress.InformationUnits	234
30	Tests	235
30.1	The Object Fortress.Standard.TestSuite	235
30.2	Test Functions	235
31	Convenience Functions and Types	236
31.1	Convenience Functions	236
31.2	Convenience Types	237

IV Fortress for Library Writers	238
32 Parallelism and Locality	239
32.1 Regions	239
32.2 Distributed Arrays	240
32.3 Abortable Atomicity	240
32.4 Shared and Local Data	241
32.5 Distributions	242
32.6 Early Termination of Threads	243
32.7 Placing Threads	244
32.8 Use and Definition of Generators	245
33 Overloaded Functional Declarations	251
33.1 Principles of Overloading	251
33.2 Subtype Rule	252
33.3 Incompatibility Rule	253
33.4 Meet Rule	253
33.5 Coercion and Overloading Resolution	255
34 Operator Declarations	256
34.1 Infix/Multifix Operator Declarations	256
34.2 Prefix Operator Declarations	257
34.3 Postfix Operator Declarations	257
34.4 Nofix Operator Declarations	257
34.5 Bracketing Operator Declarations	258
34.6 Subscripting Operator Method Declarations	258
34.7 Subscripted Assignment Operator Method Declarations	258
34.8 Conditional Operator Declarations	259
34.9 Big Operator Declarations	259
35 Dimensions and Units Declarations	260
35.1 Dimensions Declarations	260
35.2 Units Declarations	261
35.3 Abbreviating Dimension and Unit Declarations	262
35.4 Absorbing Units	263

36 Support for Domain-Specific Languages	265
36.1 Definitions of Syntax Expanders	265
36.2 Declarations of Syntax Expanders	266
36.3 Restrictions on Delimiters	266
36.4 Processing Syntax Expanders	266
36.5 Expanders for Fortress	268
V Fortress APIs and Documentation for Library Writers	269
37 Algebraic Constraints	270
37.1 Predicates and Equivalence Relations	270
37.2 Partial and Total Orders	272
37.3 Operators and Their Properties	275
37.4 Lattices	280
37.5 Monoids, Groups, Rings, and Fields	282
37.6 Boolean Algebras	287
38 Numbers	289
38.1 The Trait <code>Fortress.Standard.RationalQuantity</code>	289
38.2 The Trait <code>Fortress.Standard.TotalComparison</code>	294
38.3 Top-level Total Comparison Values	295
38.4 The Trait <code>Fortress.Standard.Comparison</code>	295
38.5 Top-level Comparison Value	296
39 Components and APIs	297
40 Memory Sequences and Binary Words	299
40.1 The Trait <code>Fortress.Core.LinearSequence</code>	300
40.2 Constructing Linear Sequences	304
40.3 The Trait <code>Fortress.Core.HeapSequence</code>	304
40.4 Constructing Heap Sequences	306
40.5 The Trait <code>Fortress.Core.BinaryWord</code>	307
40.6 The Trait <code>Fortress.Core.BinaryEndianWord</code>	311
40.7 The Trait <code>Fortress.Core.BasicBinaryOperations</code>	316
40.8 The Trait <code>Fortress.Core.BasicBinaryWordOperations</code>	320

40.9	The Trait Fortress.Core.BinaryLinearEndianSequence	322
40.10	The Trait Fortress.Core.BinaryEndianLinearEndianSequence	326
40.11	The Trait Fortress.Core.BinaryHeapEndianSequence	331
40.12	The Trait Fortress.Core.BinaryEndianHeapEndianSequence	331
40.13	The Trait Fortress.Core.BasicBinaryHeapSubsequenceOperations	332
VI	Appendices	340
A	Fortress Calculi	341
A.1	Basic Core Fortress	341
A.2	Core Fortress with Where Clauses	346
A.3	Core Fortress with Overloading	353
A.4	Acyclic Core Fortress with Field Definitions	359
B	Overloaded Functional Declarations	365
B.1	Proof of Coercion Resolution for Functions	365
B.2	Proof of Overloading Resolution for Functions	366
C	Components and APIs	368
D	Rendering of Fortress Identifiers	370
E	Support for Unicode Input in ASCII	374
E.1	Word Pasting across Line Breaks	374
E.2	Converting Names of Unicode Characters and ASCII Shorthand	375
E.3	Apostrophe Replacement within Numerals	378
E.4	Equivalence under ASCII Conversion	378
F	Operator Precedence, Chaining, and Enclosure	379
F.1	Bracket Pairs for Enclosing Operators	379
F.2	Vertical-Line Operators	380
F.3	Arithmetic Operators	381
F.4	Relational Operators	384
F.5	Boolean Operators	391
F.6	Other Operators	392

G	Simplified Grammar for Application Programmers and Library Writers	403
G.1	Components and APIs	404
G.2	Top-level Declarations	405
G.3	Trait and Object Declarations	405
G.4	Variable Declarations	407
G.5	Function Declarations	407
G.6	Dimension, Unit, Type Alias, Test, Property, and External Syntax Declarations	407
G.7	Headers	407
G.8	Parameters	409
G.9	Method Declarations	410
G.10	Method Parameters	410
G.11	Field Declarations	410
G.12	Abstract Field Declarations	411
G.13	Expressions	411
G.14	Expressions Enclosed by Keywords	412
G.15	Local Declarations	413
G.16	Literals	414
G.17	Types	414
G.18	Symbols and Operators	415
G.19	Identifiers	415
H	Full Grammar for Fortress Implementors	416
H.1	Components and APIs	416
H.2	Top-level Declarations	417
H.3	Trait and Object Declarations	418
H.4	Variable Declarations	419
H.5	Function Declarations	420
H.6	Dimension, Unit, Type Alias, Test, Property, and External Syntax Declarations	420
H.7	Headers	420
H.8	Parameters	423
H.9	Method Declarations	424
H.10	Method Parameters	424
H.11	Field Declarations	425
H.12	Abstract Field Declarations	425

H.13 Expressions	425
H.14 Expressions Enclosed by Keywords	428
H.15 Local Declarations	429
H.16 Literals	430
H.17 Expressions without Newlines	431
H.18 Expressions within Array Expressions	431
H.19 Types	432
H.20 Symbols and Operators	434
H.21 Identifiers	436
H.22 Spaces and Comments	436
I Changes between Fortress 1.0 α and 1.0 β Specifications	437

Part I

Preliminaries

Chapter 1

Introduction

The Fortress Programming Language is a general-purpose, statically typed, component-based programming language designed for producing robust high-performance software with high programmability.

In many ways, Fortress is intended to be a “growable language”, i.e., a language that can be gracefully extended and applied in new and unanticipated contexts. Fortress supports state-of-the-art compiler optimization techniques, scaling to unprecedented levels of parallelism and of addressable memory. Fortress has an extensible component system, allowing separate program components to be independently developed, deployed, and linked in a modular and robust fashion. Fortress also supports modular and extensible parsing, allowing new notations and static analyses to be added to the language.

The name “Fortress” is derived from the intent to produce a “secure Fortran”, i.e., a language for high-performance computation that provides abstraction and type safety on par with modern programming language principles. Despite this etymology, the language is a new language with little relation to Fortran other than its intended domain of application. No attempt has been made to support backward compatibility with existing versions of Fortran; indeed, many new language features were invented during the design of Fortress. Many aspects of Fortress were inspired by other object-oriented and functional programming languages, including The Java™ Programming Language [5], NextGen [6], Scala [23], Eiffel [18], Self [1], Standard ML [20], Objective Caml [16], Haskell [25], and Scheme [15]. The result is a language that employs cutting-edge features from the programming-language research community to achieve an unprecedented combination of performance and programmability.

Fortress is an open source project. An initial interpreter, implementing a core of the language features presented in this specification, is available at the Fortress project website:

`http://fortress.sunsource.net`

There you will find source code, supporting documents, and access to many discussion groups related to the Fortress project.

1.1 Fortress in a Nutshell

Two basic concepts in Fortress are that of *object* and of *trait*. An object consists of *fields* and *methods*. The fields of an object are specified in its definition. An object definition may also include method definitions.

Traits are named program constructs that declare sets of methods. They were introduced in the Self programming language, and their semantic properties (and advantages over conventional class inheritance) were analyzed by Schärli, Ducasse, Nierstrasz, and Black [8]. In Fortress, a method declared by a trait may be either *abstract* or *concrete*:

abstract methods have only *headers*; concrete methods also have *definitions*. A trait may *extend* other traits: it *inherits* the methods provided by the traits it extends. A trait provides the methods that it inherits as well as those explicitly declared in its declaration.

Every object extends a set of traits (its “supertraits”). An object inherits the concrete methods of its supertraits and must include a definition for every method declared but not defined by its supertraits.

```
object SolarSystem extends { StarSystem, OrbitingObject }
  sun = Sol
  planets = { Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune }
  position = Polar(25000 light Years, 0 radians)
   $\omega$  :  $\mathbb{R}^{64}$  AngularVelocity =  $2\pi$  radians / 226 million years in seconds
  variation( $\omega_{\Delta}$ ) =
     $\omega$  +=  $\omega_{\Delta}$ 
end
```

In this example, the object `SolarSystem` extends the traits `StarSystem` and `OrbitingObject`. The fields ω and `position` are defined with appropriate quantities. The field `sun` is defined to be another object named `Sol`, and the field `planets` is defined to be a set of objects. The method `variation` is defined to take a single parameter ω_{Δ} , and update the ω field of the object. As this example illustrates, Fortress provides static checking of physical units and dimensions on quantities.

Note that the identifiers used in this example are not restricted to ASCII character sequences. Fortress allows the use of Unicode characters [27] in program identifiers, as well as subscripts and superscripts. (See Appendix E for a discussion of Unicode and support for entering programs in ASCII.) Fortress also includes a set of standard formatting rules that follow the conventions of mathematical notation. For example, most variable references in Fortress programs are italicized. Moreover, multiplication can be expressed by simple juxtaposition, as can be seen in the definition of ω . There is also support for operator overloading, as well as a facility for extending the syntax with domain-specific languages.

Although Fortress is statically and nominally typed, types are not specified for all fields, nor for all method parameters and return values. Instead, wherever possible, *type inference* is used to reconstruct types. In the examples throughout this specification, we often omit the types when they are clear from context. Additionally, types can be parametric with respect to other types and values (most notably natural numbers).

These design decisions are motivated in part by our goal of making the scientist/programmer’s life as easy as possible without compromising good software engineering. In particular, they allow us to write Fortress programs that preserve the look of standard mathematical notation.

In addition to objects and traits, Fortress allows the programmer to define top-level functions. Functions are first-class values: They can be passed to and returned from functions, and assigned as values to fields and variables. Functions and methods can be overloaded, with calls to overloading methods resolved by multiple dynamic dispatch similarly to the manner described in [19]. Keyword parameters and variable size argument lists are also supported.

Fortress programs are organized into *components*, which export and import APIs and can be linked together. APIs describe the “shape” of a component, specifying the types in traits, objects and functions provided by a component. All external references within a component (i.e., references to traits, objects and functions implemented by other components) are to APIs imported by the component. We discuss components and APIs in detail in Chapter 22.

To address the needs of modern high-performance computation, Fortress also supports a rich set of operations for defining parallel execution and distribution of large data structures. This support is built into the core of the language. For example, `for` loops in Fortress are parallel by default.

1.2 Acknowledgments

With the help of many contributors and supporters, we are in the midst of building an open source interpreter for Fortress. Many people have contributed to the Fortress project with useful comments and suggestions for both the specification and implementation, code contributions, and supporting tools. Special thanks go to Mike Atkinson, Alex Battisti, Steve Brandt, Martin Buchholz, Zoran Budimlic, Bill Callahan, Corky Cartwright, Darren Dale, David Detlefs, Matthias Felleisen, Robby Findler, Kathi Fisler, Richard Gabriel, Markus Gaisbauer, Alex Garthwaite, Jesse Glick, Brian Goetz, Robert Grimm, Steve Heller, Maurice Herlihy, Mack Joyner, Miriam Kadansky, Kazuhiko Kakehi, Ken Kennedy, Alex Kravets, Shriram Krishnamurthi, Eric Lavigne, Doug Lea, Vass Litvinov, Eugene Loh, Daniel MD, John Mellor-Crummey, Jim Mitchell, Mark Moir, Akhilesh Mritunjai, Dan Nussbaum, Bill Pugh, Doron Rajwan, John Reppy, Nir Shavit, Bob Sproull, Walid Taha, Peter Vanderbilt, Michael Van de Vanter, Chris Vick, Larry Votta, and Jim Waldo.

We would also like to thank the anonymous referees of peer-reviewed papers related to the Fortress project, who have influenced the language design in many ways, as well as the DARPA reviewers from the HPCS project for their thoughtful feedback throughout the design of the language.

1.3 Organization

This language specification is organized as follows. In Part II, the Fortress language features for application programmers are explained, including objects, types, and functions. Relevant parts of the concrete syntax are provided with many examples. The full concrete syntax of Fortress is described in Appendix G. In Part III, APIs and documentation of some of the Fortress standard libraries for application programmers are presented. Part IV describes advanced Fortress language features for library writers and Part V presents APIs and documentation for some of the Fortress standard libraries for library writers. Finally, in Part VI, the Fortress calculi, support for Unicode characters, and the Fortress grammars are described.

A note on the presented libraries in Parts III and V: The Fortress standard libraries presented in this draft specification should not be construed as exhaustive or complete. Presentation of additional libraries is planned for future drafts, as are modifications to the libraries included here.

Chapter 2

Overview

In this chapter, we provide a high-level overview of the entire Fortress language. We present most features in this chapter through the use of examples, which should be accessible to programmers of other languages. In this chapter, unlike the rest of the specification, no attempt is made to provide complete descriptions of the various language features presented. Instead, we intend this overview to provide useful context for reading other sections of this specification, which provide rigorous definitions for what is merely introduced here.

2.1 The Fortress Programming Environment

Although Fortress is independent of the properties of a particular platform on which it is implemented, it is helpful to describe a programming model for it that we intend to provide on modern operating systems. In this programming model, Fortress source code is stored in files and organized in directories, and there is a text-based shell from which we can store environment variables and issue commands to execute and compile programs.

There are two ways in which to run a Fortress program:

- As a *script*. The Fortress program is stored in a file with the suffix “.fsx” and executed directly from an underlying operating system shell by calling the command “fortress script” on it. For example, suppose we write the following “Hello, world!” program to a file “HelloWorld.fsx”:

```
export Executable
run(args) = print “Hello, world!”
```

The first line is an *export statement*; we ignore it for the moment. The second line defines a function *run*, which takes a parameter named *args* and prints the string “Hello, world!”. Note that the parameter *args* does not include a declaration of its type. In many cases, types can be elided in Fortress and inferred from context. (In this case, the type of *args* is inferred based on the program’s export statement, explained in Section 2.2.)

We can execute this program by issuing the following command to the shell:

```
fortress script HelloWorld.fsx
```

- As a *compiled* file. In this case, the Fortress program is stored in a file with the suffix “.fss” and compiled into one or more *components*, which are stored in a persistent database called a *fortress*. Typically, a single fortress holds all the components of a user, or group of users sharing programs and libraries. In our examples, we often refer to the fortress we are issuing commands to as *the resident fortress*.

For example, we could have written our “Hello, world!” program in a compiled file “HelloWorld.fss”:

```
component HelloWorld
  export Executable
  run(args) = print "Hello, world!"
end
```

We can compile this program, by issuing the command “`fortress compile`” on it:

```
fortress compile HelloWorld.fss
```

As a result of this command, a component named “HelloWorld” is stored in the resident fortress. The name of this component is provided by the enclosing component declaration surrounding the code. If there is no enclosing component declaration, then the contents of the file are understood to belong to a single component whose name is that of the file it is stored in, minus its suffix. For example, suppose we write the following program in a source file named “HelloWorld2.fss”:

```
export Executable
run(args) = print "Hi, it's me again!"
```

When we compile this file:

```
fortress compile HelloWorld2.fss
```

the result is that a new component with the name HelloWorld2 is stored in the resident fortress. Once this component is compiled, we can execute it by issuing the following command:

```
fortress run HelloWorld2
```

In a script file, there must be at most one component declaration. In a compiled file, multiple component declarations may be included. For example, we could write the following file HelloWorld3.fss:

```
component HelloWorld
  export Executable
  run(args) = print "Hello, world!"
end
component HelloWorld2
  export Executable
  run(args) = print "Hi, it's me again!"
end
```

When we compile this file, the result is that both the components HelloWorld and HelloWorld2 are stored in the resident fortress.

If a fortress already contains a component with the same name as a newly installed component, the new component shadows the old one. For example, if we first compile the source file HelloWorld3.fss above and then compile the following file HelloWorld4.fss:

```
component HelloWorld
  export Executable
  run(args) = print "I didn't expect that!"
end
```

then executing the component HelloWorld on our fortress will result in printing of the following text:

```
I didn't expect that!
```

We can also “remove” a component from a fortress. For example:

```
fortress remove HelloWorld
```

After issuing this command, we can no longer refer to `HelloWorld` component when issuing commands to the fortress. (However, a removed component might still exist as a constituent of other, *linked*, components; see Section 2.2.)

2.2 Exports, Imports, and Linking Components

When a component is defined, it can include *export statements*. For example, all of the components we have defined thus far have included the export statement “`export Executable`”. Export statements list various *APIs* that a component implements. Unlike in other languages, APIs in Fortress are themselves program constructs; programmers can rely on standard APIs, and declare new ones. API declarations are sequences of declarations of variables, functions, and other program constructs, along with their types and other supporting declarations. For example, here is the definition of API `Executable`:

```
api Executable
  run: String... → ()
end
```

This API contains the declaration of a single function `run`, whose type is `String... → ()`. This type is an *arrow type*; it declares the type of a function’s parameter, and its return type. The function `run` includes a single parameter; the notion `String...` indicates that it is a *varargs* parameter; the function `run` can be called with an arbitrary number of string arguments. For example, here are valid calls to this function:

```
run(“a simple”, “ example”)
run(“run(...)”)
run(“Nobody”, “expects”, “that”)
```

The return type of `run` is `()`, pronounced “void”. Type `()` may be used in Fortress as a return type for functions that have no meaningful return value. There is a single value with type `()`: the value `()`, also pronounced “void”. References to value `()` as opposed to type `()` are resolved by context.

As with components, APIs can be defined in files and compiled. APIs must be defined in files with the suffix `.fsi`. An `.fsi` file contains source code for one or more APIs. If there are no explicit “`api`” headers, the file is understood to define a single API, whose name is the name of the containing file, minus its suffix.

An API is compiled with the shell command “`fortress compile`”. When an API is compiled, it is installed in the resident fortress.

For example, if we store the following API in a file named “`Blarf.fsi`”:

```
api Zeepf
  foo: String → ()
  baz: String → String
end
```

then we can compile this API with the following shell command:

```
fortress compile Blarf.fsi
```

This command compiles the API `Zeepf` and installs it in the resident fortress. If we omit the enclosing API declaration, so that the file `Blarf.fsi` consists solely of the following code:

```
foo: String → ()
baz: String → String
```

then the file is assumed to consist of the declaration of a single API named Blarf.

Unlike component compilation, API compilation does not shadow existing elements of a fortress. If we attempt to compile an API with the same name as an API already defined in the resident fortress, an error is signaled and the fortress is left unchanged. To remove an API, we must first remove all components referring to the API, and then issue the shell command “`fortress removeApi`”.

A component that exports an API must provide a definition for every program construct declared in the API. For example, because our component HelloWorld:

```
component HelloWorld
  export Executable
  run(args) = print “Hello, world!”
end
```

exports the API Executable, it must include a definition for the function `run`. The definition of `run` in HelloWorld need not include declarations of the parameter type or return type of `run`, as these can be inferred from the definition of API Executable.

Components are also allowed to *import* APIs. A component that imports an API is allowed to use any of the program constructs declared in that API. For example, the following component imports API Zeepf and calls the function `foo` declared in Zeepf:

```
component Blargh
  import Zeepf
  export Executable
  run(args) = Zeepf.foo(“whatever”)
end
```

Component Blargh imports the API Zeepf and exports the API Executable. Its `run` function is defined by calling function `foo`, defined in Zeepf. Note that `foo` must be referred to by the *qualified name* `Zeepf.foo`, to distinguish it from other declarations of `foo` that are imported by or defined in Blargh. To call `foo` as an unqualified name, we can write the following form of import statement:

```
component Blargh
  import foo from Zeepf
  export Executable
  run(args) = foo(“whatever”)
end
```

In an import statement of the form:

```
import S from A
```

all names in the set of names `S` are imported from API `A`, and can be referred to as unqualified names within the importing component. In the example above, the set of names we have imported consists of a single name: `foo`. If we had instead written:

```
import {foo, baz} from Zeepf
```

then we would have been able to refer to both *foo* and *baz* as unqualified names in Blargh.

Note that no component refers directly to another component, or to constructs defined in another component. Instead, *all external references go through APIs*. This level of indirection provides us with significant power. As we will see, it is possible to link together arbitrary components, so long as their APIs match. Programmers are able to link together components from separate programming teams, swap in revised components into deployed applications, and even test components that rely on expensive libraries by wiring them up to special *mock components* that provide just enough functionality to allow for testing.

Components that contain no import statements and export the API Executable are referred to as *executable components*. They can be compiled and executed directly as stand-alone components. All of our HelloWorld components are executable components. However, if a component imports one or more APIs, it cannot be executed as a stand-alone program. Instead, the component must be compiled and then linked with other components that export all of the APIs it imports, to form a new *compound* component. For example, we define the following component in a file named `Ralph.fss`:

```
export Zeepf
foo(s) = ()
baz(s) = s
```

We can now issue the following shell commands:

```
fortress compile Ralph.fss
fortress compile Blargh.fss
fortress link Gary from Ralph with Blargh
```

The first two commands compile files `Ralph.fss` and `Blargh.fss`, respectively, and install them in the resident fortress. The third command tells the resident fortress to link components `Ralph` and `Blargh` together into a compound component, named `Gary`. `Gary` is an executable component; we can execute it directly with the command:

```
fortress run Gary
```

All references to API `Zeepf` in `Gary` are resolved to the declarations provided in `Ralph`.

Note that forming the compound component `Gary` has no effect on the components `Ralph` and `Blargh`. These components remain in the resident fortress, and they can be linked together with other components to form yet more compound components. Conversely, if `Blargh` or `Ralph` is recompiled, deleted, or otherwise updated, there is no effect on `Gary`. Conceptually, `Gary` contains its own copies of the components `Blargh` and `Ralph`, and these copies are not corrupted by actions on other components. For this reason, we say that components in Fortress are *encapsulated*. (Of course, there are optimization tricks that a fortress can use to maintain the illusion of encapsulation without actually copying. But these tricks are beyond the scope of this specification.)

Compound components are *upgradable*: They can be upgraded with new components that export some of the APIs used by their constituents. For example, if a new version of `Ralph` is compiled and installed in the resident fortress, we can manually *upgrade* `Gary` with the new version by performing the following shell command:

```
fortress upgrade NewGary from Gary with Ralph
```

This command produces a new component, named `NewGary`, resulting from an upgrade of `Gary` with `Ralph`. The components referred to as `Gary` and `Ralph` are unaffected. An important property of fortress components is that they are *stateless*; once constructed, they are never modified. Even if a component is “removed” from a fortress, the components it is a constituent of are unaffected.

However, we can rebind *the name* `Gary` in the resident fortress to our new component with the following command:

```
fortress upgrade Gary from Gary with Ralph
```

or simply:

```
fortress upgrade Gary with Ralph
```

Now, the original component referred to by `Gary` is shadowed; it cannot be referred to directly from the shell. However, it might still exist in the fortress as a constituent of other components, which are unmodified by the upgrade. To upgrade all components in a fortress at once with a new version of `Ralph` (rebinding all names to the resulting upgrades), we can issue the command:

```
fortress upgradeAll Ralph
```

2.3 Automatic Generation of APIs

Note that the component named `Zeepf` exports the API `Zeepf`. Components and APIs exist in separate namespaces, and therefore it is allowed for a component to have the same name as an API. In fact, if we hadn't already defined and compiled API `Zeepf`, we could generate it automatically from the definition of component `Zeepf` with the following shell command:

```
fortress api Zeepf.fss
```

This command generates a new source file, `Zeepf.fsi`, in the same directory as `Zeepf.fss`, which includes declarations for all program constructs defined in `Zeepf.fss`.

In general, if a component `C` exports an API `A` with the same name as `C`, it is possible to automatically generate a source file for the API `A` from `C` by issuing the shell command `fortress api` on the file containing the definition of `C`. This API contains declarations for all definitions in `C` that are not declared in other APIs exported by `C`, and that do not include the modifier `private`. If a source file with the name of `A` already exists in the same directory as the source file of `C`, an error is signaled, and no file is created.

If there is more than one component defined in a file, APIs are generated for all components defined in the file that export APIs with the same name as the respective component. Each generated API is placed in a separate source file whose name corresponds to the name of the API it defines.

Note that the API corresponding to an automatically generated source file is not automatically added to the resident fortress; the source file must still be compiled via a separate action. Often, programmers may want to edit the auto-generated file before compiling it to the fortress.

It is not always desirable for a component to export an API of the same name. Many components will export only publicly defined standard APIs. However, automatic generation of APIs from components may be useful, particularly for components defined internally by a development team. Large projects may contain many internally defined components that are never released externally, except as constituents of compound components.

2.4 Rendering

One aspect of Fortress that is quite different from other languages is that various program constructs are rendered in particular fonts, so as to emulate mathematical notation. In the examples above, this was evident by the use of italics when rendering variable names. Many other program constructs have their own rendering rules. For example, the operator `^` indicates superscripting in Fortress. A function definition consisting of the following ASCII characters:

```
f(x) = x^2 + sin x - cos 2 x
```

is rendered as follows:

$$f(x) = x^2 + \sin x - \cos 2x$$

Array indexing, written with brackets:

`a[i]`

is rendered as follows:

a_i

There are many other examples of special rendering conventions.

Fortress also supports the use of Unicode characters [27] in identifiers. In order to make it easy to enter such characters with today’s input devices, Fortress defines a convention for keyboard entry: ASCII names (and abbreviations) of Unicode characters can be written in a program in all caps (with spaces replaced by underscores). Such identifiers are converted to Unicode characters. For example, the identifier:

`GREEK_CAPITAL_LETTER_LAMBDA`

is automatically converted into the identifier:

Λ

There are also ASCII abbreviations for writing down commonly used Fortress characters. For example, ASCII identifiers for all Greek letters are converted to Greek characters (e.g., “lambda” becomes λ and “LAMBDA” becomes Λ). Here are some other common ASCII shorthands:

<code>BY</code>	<i>becomes</i>	\times	<code>TIMES</code>	<i>becomes</i>	\times
<code>DOT</code>	<i>becomes</i>	\cdot	<code>CROSS</code>	<i>becomes</i>	\times
<code>CUP</code>	<i>becomes</i>	\cup	<code>CAP</code>	<i>becomes</i>	\cap
<code>BOTTOM</code>	<i>becomes</i>	\perp	<code>TOP</code>	<i>becomes</i>	\top
<code>SUM</code>	<i>becomes</i>	\sum	<code>PRODUCT</code>	<i>becomes</i>	\prod
<code>INTEGRAL</code>	<i>becomes</i>	\int	<code>EMPTYSET</code>	<i>becomes</i>	\emptyset
<code>SUBSET</code>	<i>becomes</i>	\subset	<code>NOTSUBSET</code>	<i>becomes</i>	$\not\subset$
<code>SUBSETEQ</code>	<i>becomes</i>	\subseteq	<code>NOTSUBSETEQ</code>	<i>becomes</i>	$\not\subseteq$
<code>EQUIV</code>	<i>becomes</i>	\equiv	<code>NOTEQUIV</code>	<i>becomes</i>	$\not\equiv$
<code>IN</code>	<i>becomes</i>	\in	<code>NOTIN</code>	<i>becomes</i>	\notin
<code>LT</code>	<i>becomes</i>	$<$	<code>LE</code>	<i>becomes</i>	\leq
<code>GT</code>	<i>becomes</i>	$>$	<code>GE</code>	<i>becomes</i>	\geq
<code>EQ</code>	<i>becomes</i>	$=$	<code>NE</code>	<i>becomes</i>	\neq
<code>AND</code>	<i>becomes</i>	\wedge	<code>OR</code>	<i>becomes</i>	\vee
<code>NOT</code>	<i>becomes</i>	\neg	<code>XOR</code>	<i>becomes</i>	\oplus
<code>INF</code>	<i>becomes</i>	∞	<code>SQRT</code>	<i>becomes</i>	$\sqrt{\quad}$

A comprehensive description of ASCII conversion is provided in Appendix E.

2.5 Some Common Types in Fortress

Fortress provides a wide variety of standard types, including String, Boolean, and various numeric types. The floating-point type \mathbb{R} (written in ASCII as `RR`) is the type of 64-bit precision floating-point numbers. For example, the following function takes a 64-bit float and returns a 64-bit float:

$halve(x : \mathbb{R}) : \mathbb{R} = x/2$

The type \mathbb{R} is also written \mathbb{R}_{64} . The following definition of *halve* is semantically equivalent to the one above:

$halve(x : \mathbb{R}64) : \mathbb{R}64 = x/2$

Predictably, 32-bit precision floats are written $\mathbb{R}32$.

64-bit integers are denoted by the type $\mathbb{Z}64$. 32-bit integers by the type $\mathbb{Z}32$, and infinite precision integers by the type \mathbb{Z} .

2.6 Functions in Fortress

Fortress allows recursive, and mutually recursive function definitions. Here is a simple definition of a *factorial* function in Fortress:

```
factorial(n) =  
  if n = 0 then 1  
  else n factorial(n - 1) end
```

2.6.1 Juxtaposition and function application

In the definition of *factorial*, note the juxtaposition of parameter n with the recursive call $factorial(n - 1)$. In Fortress, as in mathematics, multiplication is represented through juxtaposition. By default, two expressions of numeric type that are juxtaposed represent a multiplication. On the other hand, juxtaposition of an expression of function type with another expression to its right represents function application, as in the following example:

$\sin x$

Moreover, juxtaposition of expressions of string type represents concatenation, as in the following example:

```
“Hi,” “ it’ s” “ me” “ again.”
```

In fact, juxtaposition is an operator in Fortress, just like $+$ and $-$, that is overloaded based on the runtime types of its arguments.

There is a special case concerning juxtaposition that is important to mention: When writing a numeric literal, the *digit-group separator*, `NARROW NO-BREAK SPACE` (U+202F), may be included between the digits in order to improve readability. In ASCII, this character can be abbreviated in the context of a numeric literal (*and only in this context*) with the character `'`. For example, the number 299,792,458 can be denoted in ASCII by the following numeric literal:

```
299'792'458
```

This literal is converted into Unicode by replacing all occurrences of `'` with `NARROW NO-BREAK SPACE`:

```
299 792 458
```

This is not an application of the juxtaposition operator; rather, it is a single token. Note that commas cannot be used to separate digits because ambiguities would arise with the use of commas in other expressions, such as tuples.

2.6.2 Keyword Parameters

Functions in Fortress can be defined to take keyword arguments by providing default values for parameters. In the following example:

```

makeColor(red : ℤ64 = 0, green : ℤ64 = 0, blue : ℤ64 = 0) =
  if 0 ≤ red ≤ 255 ∧ 0 ≤ green ≤ 255 ∧ 0 ≤ blue ≤ 255
  then Color(red, green, blue)
  else throw Error end

```

the function `makeColor` takes three keyword arguments, all of which default to 0. If we call it as follows:

```
makeColor(green = 255)
```

the arguments `red` and `blue` are both given the value 0.

There are some other aspects of this example worth mentioning. For example, the body of this function consists of an `if` expression. The test of the `if` expression checks that all three parameters have values between 0 and 255. The boolean operator `≤` is *chained*: An expression of the form $x \leq y \leq z$ is equivalent to the expression $x \leq y \wedge y \leq z$, just as in mathematical notation. The `then` clause provides an example of a constructor call in Fortress, and the `else` clause shows us an example of a `throw` expression in Fortress.

2.6.3 Varargs Parameters

It is also possible to define functions that take a variable number of arguments. We have already seen such a function: `Executable.run`. Here is another:

```

printFirst(xs : ℝ...) =
  if |xs| > 0 then print xs_0
  else throw Error end

```

This function takes an arbitrary number of floats and prints the first one (unless it is given zero arguments; then it throws an exception).

2.6.4 Function Overloading

Functions can be overloaded in Fortress by the types of their parameters. Calls to overloaded functions are resolved based on the runtime types of the arguments. For example, the following function is overloaded based on parameter type:

```

size(x : Nil) = 0
size(x : Cons) = 1 + size(rest(x))

```

If we call `size` on an object with runtime type `Cons`, the second definition of `size` will be invoked *regardless of the static type of the argument*. Of course, function applications are statically checked to ensure that *some* definition will be applicable at run time, and that the definition to apply will be unambiguous.

2.6.5 Function Contracts

Fortress allows *contracts* to be included in function declarations. Among other things, contracts allow us to *require* that the argument to a function satisfies a given set of constraints, and to *ensure* that the resulting value satisfies some constraints. They provide essential documentation for the clients of a function, enabling us to express semantic properties that cannot be expressed through the static type system.

Contracts are placed at the end of a function header, before the function body. For example, we can place a contract on our `factorial` function requiring that its argument be nonnegative as follows:

```
factorial(n) requires { n ≥ 0 }
= if n = 0 then 1
  else n factorial(n - 1) end
```

We can also ensure that the result of *factorial* is itself nonnegative:

```
factorial(n)
  requires { n ≥ 0 }
  ensures { result ≥ 0 }
= if n = 0 then 1
  else n factorial(n - 1) end
```

The variable *result* is bound in the `ensures` clause to the return value of the function.

2.7 Some Common Expressions in Fortress

We have already seen an `if` expression in Fortress. Here's an example of a `while` expression:

```
while x < 10 do
  print x
  x += 1
end
```

Blocks in Fortress are delimited by `do` and `end`. Here is an example of a function that prints three words:

```
printThreeWords() = do
  print "print"
  print "three"
  print "words"
end
```

A *tuple* expression contains a sequence of elements delimited by parentheses and separated by commas:

```
("this", "is", "a", "tuple", "of", "mostly", "strings", 0)
```

When a tuple expression is evaluated, the various subexpressions are evaluated *in parallel*. For example, the following tuple expression denotes a parallel computation:

```
(factorial(100), factorial(500), factorial(1000))
```

The elements in this expression may be evaluated in parallel. This same computation can be expressed in *block form* as follows:

```
do
  factorial(100)
also do
  factorial(500)
also do
  factorial(1000)
end
```

2.8 For Loops Are Parallel by Default

Here is an example of a simple `for` loop in Fortress:

```
for i ← 1 : 10 do
  print(i " ")
end
```

This `for` loop iterates over all elements i between 1 and 10 and prints the value of i . Expressions such as `1 : 10` are referred to as *range expressions*. They can be used in any context where we wish to denote all the integers between a given pair of integers.

A significant difference between Fortress and most other programming languages is that *for loops are parallel by default*. Thus, printing in the various iterations of this loop can occur in an arbitrary order, such as:

```
5 4 6 3 7 2 9 10 1 8
```

2.9 Atomic Expressions

In order to control interactions of parallel executions, Fortress includes the notion of *atomic expressions*, as in the following example:

```
atomic do
  x += 1
  y += 1
end
```

An atomic expression is executed in such a manner that all other threads observe either that the computation has completed, or that it has not yet begun; no other thread observes an atomic expression to have only partially completed. Consider the following parallel computation:

```
do
  x:ℤ := 0
  y:ℤ := 0
  z:ℤ := 0
  atomic do
    x += 1
    y += 1
  also atomic do
    z := x + y
  end
  z
end
```

Both parallel blocks are atomic; thus the second parallel block either observes that both x and y have been updated, or that neither has. Thus, possible values of the outermost `do` expression are 0 and 2, but not 1.

2.10 Dimensions and Units

Numeric types in Fortress can be annotated with physical units and dimensions. For example, the following function declares that its parameter is a tuple represented in the units `kg` and `m/s`, respectively:

$$\textit{kineticEnergy}(m : \mathbb{R} \text{ kg}, v : \mathbb{R} \text{ m/s}) : \mathbb{R} \text{ kg m}^2/\text{s}^2 = (m v^2)/2$$

A value of type $\mathbb{R} \text{ kg}$ is a 64-bit float representing a measurement in kilograms. When the function *kineticEnergy* is called, two values in its tuple argument are statically checked to ensure that they are in the right units.

All commonly used dimensions and units are provided in the Fortress standard libraries. Unit symbols are encoded with trailing underscores; such identifiers are rendered in roman font. For example the unit m is represented as $\text{m}_$. For each unit, both longhand and shorthand names are provided (e.g., m , meter , and meters). The various names of a given unit can be used interchangeably. Also, some units (and dimensions) are defined to be synonymous with algebraic combinations of other units (and dimensions). For example, the unit N is defined to be synonymous with the unit kg m/s^2 and the dimension Force is defined to be synonymous with Mass Acceleration . Likewise, the dimension Acceleration is defined to be synonymous with Velocity/Time .

Measurements in the same unit can be compared, added, subtracted, multiplied and divided. Measurements in different units can be multiplied and divided. For example, we can write the following variable declaration:

$$v : \mathbb{R} \text{ m/s} = (3 \text{ meters} + 4 \text{ meters})/5 \text{ seconds}$$

However, the following variable declaration is a static error:

$$v : \mathbb{R} \text{ m/s} = (3 \text{ meters} + 4 \text{ seconds})/5 \text{ seconds}$$

In addition, the following variable declaration is a static error because the unit of the left hand side of this declaration is m/s whereas the unit of the right hand side is simply m (or meters):

$$v : \mathbb{R} \text{ m/s} = (3 \text{ meters} + 4 \text{ meters})/5$$

It is also possible to convert a measurement in one unit to a measurement of another unit of the same dimension, as in the following example:

$$\textit{kineticEnergy}(3.14 \text{ kg}, 32 \text{ f/s in m/s})$$

The second argument to *kineticEnergy* is a measurement in feet per second, converted to meters per second.

2.11 Aggregate Expressions

As with mathematical notation, Fortress includes special syntactic support for writing down many common kinds of collections, such as tuples, arrays, matrices, vectors, maps, sets, and lists simply by enumerating all of the collection's elements. We refer to an expression formed by enumerating the elements of a collection as an *aggregate expression*. The elements of an aggregate expression are computed in parallel.

For example, we can define an array a in Fortress by explicitly writing down its elements, enclosed in brackets and separated by whitespace, as follows:

$$a = [0 \ 1 \ 2 \ 3 \ 4]$$

Two-dimensional arrays can be written down by separating rows by newlines (or by semicolons). For example, we can bind b to a two-dimensional array as follows:

$$b = \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix}$$

There is also support for writing down arrays of dimension three and higher. We bind c to a three-dimensional array as follows:

$$c = [1\ 2 \\ 3\ 4; 5\ 6 \\ 7\ 8; 9\ 10 \\ 11\ 12]$$

Various slices of the array along the third dimension are separated by pairs of semicolons. (Higher dimensional arrays are also supported. When writing a four-dimensional array, slices along the fourth dimension are separating by triples of semicolons, and so on.)

Vectors are written down just like one-dimensional arrays. Similarly, matrices are written down just like two-dimensional arrays. Whether an array aggregate expression evaluates to an array, a vector, or a matrix is inferred from context (e.g., the static type of a variable that such an expression is bound to). Of course, all elements of vectors and matrices must be numbers.

Note that there is a syntactic conflict between the use of juxtaposition to represent elements along a row in an array and the use of juxtaposition to represent applications of the juxtaposition operator. In order to include an application of the juxtaposition operator as an outermost subexpression of an array aggregate, it is necessary to include extra parentheses. For example, the following aggregate expression represents a counterclockwise rotation matrix by an angle θ in \mathbb{R}^2 :

$$[(\cos \theta)(-\sin \theta) \\ (\sin \theta)(\cos \theta)]$$

A set can be written down by enclosing its elements in braces and separating its elements by commas. Here we bind s to the set of integers 0 through 4:

$$s = \{0, 1, 2, 3, 4\}$$

The elements of a list are enclosed in angle brackets (written in ASCII as `<|` and `|>`):

$$l = \langle 0, 1, 2, 3, 4 \rangle$$

The elements of a map are enclosed in curly braces, with key/value pairs joined by the arrow \mapsto (written in ASCII as `|->`):

$$m = \{ 'a' \mapsto 0, 'b' \mapsto 1, 'c' \mapsto 2 \}$$

2.12 Comprehensions

Another way in which Fortress mimics mathematical notation is in its support for *comprehensions*. Comprehensions describe the elements of a collection by providing a rule that holds for all of the collection's elements. The elements of the collection are computed in parallel by default. For example, we define a set s that consists of all elements of another set t divided by 2, as follows:

$$s = \{x/2 \mid x \leftarrow t\}$$

The expression to the left of the vertical bar explains that elements of s consist of every value $x/2$ for every valid value of x (determined by the right hand side). The expression to the right of the vertical bar explains how the elements x are to be *generated* (in this case, from the set t). The right hand side of a comprehension can consist of multiple generators. For example, the following set consists of every element resulting from the sum of an element of s with an element of t :

$$u = \{x + y \mid x \leftarrow s, y \leftarrow t\}$$

The right hand side of a comprehension can also contain *filtering expressions* to constrain the elements provided by other clauses. For example, we can stipulate that v consists of all nonnegative elements of t as follows:

$$v = \{x \mid x \leftarrow t, x \geq 0\}$$

There is a comprehension expression for every form of aggregate expression except tuple expressions. For example, here is a list comprehension in Fortress:

$$\langle 2x \mid x \leftarrow v \rangle$$

The elements of this list consist of all elements of the set v multiplied by 2.

In the case of an array comprehension, the expression to the left of the bar includes a tuple indexing the elements of the array. For example, the following comprehension describes a 3×3 array, all of whose elements are zero.

$$[(x, y) = 0 \mid x \leftarrow \{0, 1, 2\}, y \leftarrow \{0, 1, 2\}]$$

The collection of elements 0 through 2 can also be expressed via a range expression, as follows:

$$[(x, y) = 0 \mid x \leftarrow 0:2, y \leftarrow 0:2]$$

An array comprehension can consist of multiple clauses. The clauses are run in the order provided, with declarations from later clauses shadowing declarations from earlier clauses. For example, the following comprehension describes a 3×3 identity matrix:

$$\begin{aligned} [(x, y) = 0 \mid x \leftarrow 0:2, y \leftarrow 0:2 \\ (x, x) = 1 \mid x \leftarrow 0:2] \end{aligned}$$

2.13 Summations and Products

As with mathematical notation, Fortress provides syntactic support for summations and productions (and other big operations) over the elements of a collection. For example, an alternative definition of *factorial* is as follows:

$$factorial(n) = \prod_{i \leftarrow 1:n} i$$

This function definition can be written in ASCII as follows:

```
factorial(n) = PRODUCT[i <- 1:n] i
```

The character \prod is written `PRODUCT`. Likewise, \sum is written `SUM`. As with comprehensions, the values in the iteration are generated from specified collections.

2.14 Tests and Properties

Fortress includes support for automated program testing. New tests in a component can be defined using the `test` modifier. For example, here is a test that calls to the *factorial* function result in values greater than or equal to what was provided:

```
test factorialResultLarger[x <- 0:100] = x ≤ factorial(x)
```

The values for x are generated from the provided range $0 : 100$.

Programs are also allowed to include *property* declarations, documenting boolean conditions that a program is expected to obey. Property declarations are similar syntactically to test declarations. Unlike test declarations, property declarations do not specify explicit finite collections over which the property is expected to hold. Instead, the parameters in a property declaration are declared to have types, and the property is expected to hold over all values of those types. For example, here is a property declaring that *factorial* is greater than or equal to every argument of type \mathbb{Z} :

```
property factorialResultAlwaysLarger =  $\forall(x : \mathbb{Z}) (x \leq \text{factorial}(x))$ 
```

When a property declaration includes a name, the property identifier is bound to a function whose parameter and body are that of the property, and whose return type is Boolean. A function bound in this manner is referred to as a *property function*. A property function can be called from within tests to ensure that a property holds at least for all values in some finite set of values.

2.15 Objects and Traits

A great deal of programming can be done simply through the use of functions, top-level variables, and standard types. However, Fortress also includes a trait and object system for defining new types, as well as objects that belong to them. Traits in Fortress exist in a multiple inheritance hierarchy rooted at trait `Object`. A trait declaration includes a set of method declarations, some of which may be abstract. For example, here is a declaration of a simple trait named `Moving`:

```
trait Moving extends { Tangible, Object }
  position(): ( $\mathbb{R}$  Length)3
  velocity(): ( $\mathbb{R}$  Velocity)3
end
```

The set of traits extended by trait `Moving` are listed in braces after `extends`. Trait `Moving` inherits all methods declared by each trait it extends. The two methods `position` and `velocity` declared in trait `Moving` are *abstract*; they contain no body. Their return types are vectors of length 3, whose elements are of types \mathbb{R} Length and \mathbb{R} Velocity respectively. As in mathematical notation, a vector of length n with element type T is written T^n .

Traits can also declare concrete methods, as in the following example:

```
trait Fast extends Moving
  velocity() = [0 0 (299 792 458 m/s)]
end
```

Trait `Fast` extends a single trait, `Moving`. (Because it extends only one trait, we can elide the braces in its `extends` clause.) It inherits both abstract methods defined in `Moving`, and it provides a concrete body for method `velocity`.

Trait declarations can be extended by other trait declarations, as well as by *object declarations*. There are two kinds of object declarations: *singleton declarations* and *constructor declarations*.

A singleton declaration declares a sole, stand-alone, *singleton object*. For example:

```
object Sol extends { Moving, Stellar }
  spectralClass =  $G_2$ 
  position() = [0 0 0]
  velocity() = [0 0 0]
end
```


The object `Sol` extends two traits: `Moving` and `Stellar`, and provides definitions for the abstract methods it inherits. Objects must provide concrete definitions for all abstract methods they inherit. `Sol` also defines a field `spectralClass`. For every field included in an object definition, an implicit getter is defined for that field. Calls to the getter of the field of an object consist of an expression denoting the object followed by a dot and the name of the field. For example, here is a call to the getter `spectralClass` of object `Sol`:

```
Sol.spectralClass
```

If a field includes modifier `settable`, an implicit setter is defined for the field. Calls to setters look like assignments. Here is an assignment to field `spectralClass` of object `Sol`:

```
Sol.spectralClass := G3
```

In fact, all accesses to a field from outside the object to which the field belongs must go through the getter of the field, and all assignments to it must go through the setter. (There is simply no other way syntactically to refer to the field.)

If a field includes modifier `hidden`, no implicit getter is defined for the object.

Every method declared in an object or trait includes an implicit `self` parameter, denoting the receiver of the method call. If desired, this parameter can be made explicit, by including it before the name of the method, along with a trailing dot. For example, the following definition of `Sol` is equivalent to the one above:

```
object Sol extends {Moving, Stellar}
  spectralClass = G2
  self.position() = [0 0 0]
  velocity() = [0 0 0]
end
```

In this definition, the `self` parameter of `position` is provided explicitly.

In fact, the `self` parameter of a method can even be given a position other than the default position. For example, here is a definition of a type where the `self` parameters appear in nonstandard positions:

```
trait List
  cons(x: Object, self): List
  append(xs: List, self): List
end
```

In both methods `cons` and `append`, the `self` parameter occurs as the second parameter. Calls to these methods look more like function calls than method calls. For example, in the following call to `append`, the receiver of the call is `l2`:

```
append(l1, l2)
```

A constructor declaration declares an *object constructor*. In contrast to singleton declarations, a constructor declaration includes value parameters in its header, as in the following example:

```
object Particle(position: (ℝ Length)3,
               velocity: (ℝ Velocity)3)
  extends Moving
end
```

Every call to the constructor of `Particle` yields a new object. For example:

```
p1 = Particle([3 2 5]m, [1 0 0]m/s)
```

The parameters to an object constructor implicitly define fields of the object, along with their getters (by default). These parameters can include all of the modifiers that an ordinary field can. For example, a parameter can include the modifier `hidden`, in which case a getter is not implicitly defined for the field.

In the definition of `Particle`, it is the implicitly defined getters of fields `position` and `velocity` that provide concrete definitions for the inherited abstract methods from trait `Moving`.

In both singleton and constructor declarations, implicitly defined getters and setters can be overridden by defining explicit getters and setters, using the modifiers `getter` and `setter`. For example, we alter the definition of `Particle` to include an explicit getter for field `velocity` as follows:

```
object Particle(position : (ℝ Length)3,
               velocity : (ℝ Velocity)3)
  extends Moving
  getter velocity() = do
    print "velocity getter accessed"
    velocity
  end
end
```

The explicitly defined getter first prints a message and then returns the value of the field `velocity`. *Note that the final variable reference in this getter refers directly to the field `velocity`, not to the getter.* Only within an object definition can fields be accessed directly. In fact, even in an object definition, fields are only accessed directly when they are referred to as simple variables. In the following definition of `Particle`, the getter `velocity` is recursively accessed through the special variable `self` (bound to the receiver of the method call):

```
object Particle(position : (ℝ Length)3,
               velocity : (ℝ Velocity)3)
  extends Moving
  getter velocity() = do
    print "velocity getter"
    self.velocity
  end
end
```

Now, the result of a call to getter `velocity` is an infinite loop (along with a lot of output).

2.15.1 Traits, Getters, and Setters

Although traits do not include field declarations, they can include getter and setter declarations, as in the following alternative definition of trait `Moving`:

```
trait Moving extends { Tangible, Object }
  getter position(): (ℝ Length)3
  getter velocity(): (ℝ Velocity)3
end
```

Now, an object extending trait `Moving` must provide the definitions of *getters* (as opposed to ordinary methods) for `position` and `velocity`. The advantage of this definition is that we can use getter notation on a variable of static type `Moving`:

```
v : Moving = ...
v.position
```

In fact, getters can be declared in a trait definition using field declaration syntax, as in the following example (which is equivalent to the definition of `Moving` above):

```
trait Moving extends { Tangible, Object }
  position: (ℝ Length)3
  velocity: (ℝ Velocity)3
end
```

Getter declarations in trait definitions must not include bodies. A getter declaration can include the various modifiers allowed on a field declaration, with similar results. For example, if the modifier `settable` is used, then a setter is implicitly declared, in addition to a getter.

2.16 Features for Library Development

The language features introduced above are sufficient for the vast majority of applications programming. However, Fortress has also designed to be a good language for *library* programming. In fact, much of the Fortress language as viewed by applications programmers actually consists of code defined in libraries, written in a small core language. By defining as much of the language as possible in libraries, our aim is to allow the language to evolve gracefully as new demands are placed on it. In this section, we briefly mention some of the features that make Fortress a good language for library development.

2.16.1 Generic Types and Static Parameters

As in other languages, Fortress allows types to be parametric with respect to other types, as well as other “static” parameters, including integers, booleans, dimensions, units, operators, and identifiers. We have already seen some standard parametric types, such as `Array` and `Vector`. Programmers can also define new traits, objects, and functions that include static parameters.

2.16.2 Specification of Locality and Data Distribution

Fortress includes a facility for expressing programmer intent concerning the distribution of large data structures at run time. This intent is expressed through special data structures called *distributions*. In fact, all arrays and matrices include distributions. These distributions are defined in the Fortress standard libraries. In most cases, the default distributions will provide programmers with good performance over a variety of machines. However, in some circumstances, performance can be improved by overriding the default selection of distributions. Moreover, some programmers might wish to define new distributions, tailored for particular platforms and programs.

2.16.3 Operator Overloading

Operators in Fortress can be overloaded with new definitions. Here is an alternative definition of the *factorial* function, defined as a postfix operator:

$$\text{opr } (n)! = \prod_{i \leftarrow 1:n} i$$

As with mathematical notation, Fortress allows operators to be defined as prefix, postfix, and infix. More exotic operators can even be defined as subscripting (i.e., applications of the operators look like subscripts), and as bracketing (i.e., applications of the operators look like the operands have been simply enclosed in brackets).

2.16.4 Definition of New Syntax

Fortress provides a facility for the definition of new syntax in libraries. This facility is useful for defining libraries for domain-specific languages, such as SQL, XML, etc. It is also used to encode some of the language constructs seen by applications programmers.

Part II

Fortress for Application Programmers

Chapter 3

Programs

A *program* consists of a finite sequence of Unicode 5.0 abstract characters. (Fortress does not distinguish between different code-point encodings of the same character.) In order to more closely approximate mathematical notation, certain sequences of characters are rendered as subscripts or superscripts, italicized or boldface text, or text in special fonts, according to the rules in Appendix E. For the purposes of entering program text, ASCII encodings of Unicode characters are also provided in Appendix E. Although much of the program text in this specification is rendered as formatted Unicode, some text is presented unformatted, or in its ASCII encoding, to aid in exposition.

A program is *valid* if it satisfies all *static constraints* stipulated in this specification. Failure to satisfy a static constraint is a *static error*. Only valid programs can be *executed*; the validity of a program must be checked before it is executed.

Executing a valid Fortress program consists of *evaluating expressions*. Evaluation of an expression may modify the *program state* yielding a *result*. A result is either a *value*, or an *abrupt completion*.

The characters of a valid program determine a sequence of *input elements*. In turn, the input elements of a program determine the *program constructs*. Some program constructs may contain other program constructs. The two most common kinds of constructs in Fortress are *declarations* and *expressions*. We explain the structure of input elements, and of each program construct, in turn, along with accompanying static constraints. We also explain how the outcome of a program execution is determined from the sequence of constructs in the program.

Programs are developed, compiled, and deployed as *encapsulated upgradable components* as described in Chapter 22.

Fortress is block-structured: A Fortress program consists of nested *blocks* of code. The entire program is a single block. Each component is a block. Any top-level declaration is a block, as is any function declaration. Several expressions are also blocks, or have blocks as parts of the expression, or both (e.g., a `while` expression is a block, and its body is a different block). See Chapter 13 for a discussion of expressions in Fortress. In addition, a local declaration begins a block that continues to the end of the smallest enclosing block unless it is a local function declaration and is immediately preceded by another local function declaration. (This exception allows overloaded and mutually recursive local function declarations.) Because Fortress is block-structured, and because the entire program is a block, the smallest block that syntactically contains a program construct is always well defined.

Fortress is expression-oriented: What are often called “statements” in other languages are just expressions with type `()` in Fortress. They do not evaluate to an interesting value, and are typically evaluated solely for their effects.

Fortress is whitespace-sensitive: Fortress has different contexts influencing the whitespace-sensitivity of expressions as described in Appendix G.

Chapter 4

Evaluation

The state of an executing Fortress program consists of a set of *threads*, and a *memory*. Communication with the outside world is accomplished through *input and output actions*. In this chapter, we give a general overview of the execution process and introduce terminology used throughout this specification to describe the behavior of Fortress constructs.

Executing a Fortress program consists of *evaluating* the body expression of the *run* function and the initial-value expressions of all top-level variables and singleton object fields in parallel. All initializations must complete normally before any reference to the objects or variables being initialized.

Threads evaluate expressions by taking *steps*. We say evaluation of an expression *begins* when the first step is taken. A step may *complete* the evaluation, in which case no more steps are possible on that expression, or it may result in an *intermediate expression*, which requires further evaluation. *Dynamic program order* is a partial order among the expressions evaluated in a particular execution of a program. When two expressions are ordered by dynamic program order, the first must complete execution before any step can be taken in the second. Chapter 13 gives the dynamic program order constraints for each expression in the language.

Intermediate expressions are generalizations of ordinary Fortress expressions: some intermediate expressions cannot be written in programs. We say that one expression is *dynamically contained* within a second expression if all steps taken in evaluating the first expression are taken between the beginning and completion of the second. A step may also have effects on the program state beyond the thread taking the step, for example, by modifying one or more locations in memory, creating new threads to evaluate other expressions, or performing an input or output action. Threads are discussed further in Section 4.4. The memory consists of a set of *locations*, which can be *read* and *written*. New locations may also be *allocated*. The memory is discussed further in Section 4.3. Finally, *input actions* and *output actions* are described in Section 4.6.

4.1 Values

A *value* is the result of normal completion of the evaluation of an expression. (See Section 4.2 for a discussion of completion of evaluation.) A value has a *type*, an *environment* (see Section 4.5), and a finite set of *fields*. Every value is an object (see Chapter 10); it may be a *value object*, a *reference object*, or a *function*. See Chapter 8 for a description of the types corresponding to these different values. The type of a value specifies the names and types of its fields, and which names must be bound in its environment (and the types of the locations they are bound to). The type also specifies the methods (including their bodies) of the object. Only trait types have methods other than those inherited from the type `Any`.

In a value object, each field is a value. In a reference object, each field is a location. Every field has a name, which may be an identifier or an index. Only values of type `LinearSequence` (defined in Section 40.1) or `HeapSequence`

(defined in Section 40.3) have fields named by indices. Functions and the value `()` have no fields. Every field in a value object is *immutable*. Reference objects may have both *mutable* and *immutable* fields. No two distinct values share any mutable field.

Values are constructed by top-level function declarations (see Section 12.1) and singleton declarations (see Section 10.1), and by evaluating an object expression (see Section 13.9), a function expression (see Section 13.7), a local function declaration (see Section 6.4), a call to an object constructor (declared by a constructor declaration; see Chapter 10), a literal (see Section 13.1), a `spawn` expression (see Section 13.23), an aggregate expression (see Section 13.27), or a comprehension (see Section 13.28). In the latter case, the constructed value is the result of the normal completion of such an evaluation.

4.2 Normal and Abrupt Completion of Evaluation

Conceptually, an expression is evaluated until it *completes*. Evaluation of an expression may *complete normally*, resulting in a value, or it may *complete abruptly*. Each abrupt completion has an associated value, either an exception value that is thrown and uncaught or the exit value of an `exit` expression (described in Section 13.12). In addition to programmer-defined exceptions thrown explicitly by a `throw` expression (described in Section 13.24), there are predefined exceptions thrown by the Fortress standard libraries. For example, dividing an integer by zero (using the `/` operator) causes a `DivideByZeroException` to be thrown.

When an expression completes abruptly, control passes to the dynamically immediately enclosing expression. This continues until the abrupt completion is handled either by a `try` expression (described in Section 13.25) if an exception is being thrown or by an appropriately tagged `label` expression (described in Section 13.12) if an `exit` expression was evaluated. If abrupt completion is not handled within a thread and its outermost expression completes abruptly, the thread itself completes abruptly. If the main thread of a program completes abruptly, the program as a whole also completes abruptly.

4.3 Memory and Memory Operations

In this specification, the term *memory* refers to a set of abstract *locations*; the memory is used to model sharing and mutation. A location has an associated type, and *contains* a value of that type (i.e., the type of the value is a subtype of the type of the location). Locations can have non-object trait types; by contrast, a value always has an object type.

There are three operations that can be performed on memory: *allocation*, *read*, and *write*. Reads and writes are collectively called *memory accesses*. Intuitively, a read operation takes a location and returns the value contained in that location, and a write operation takes a location and a value of the location's type and changes the contents of the location so the location contains the given value. Accesses need not take place in the order in which they occur in the program text; a detailed account of memory behavior appears in Chapter 21.

Allocation creates a new location of a given type. Allocation occurs when a mutable variable is declared, or when a reference object is constructed. In the latter case, a new location is allocated for each field of the object. Abstractly, locations are never reclaimed; in practice, memory reclamation is handled by garbage collection.

A freshly allocated location is *uninitialized*. The type system and memory model in Fortress guarantee that an *initializing write* is performed to every location if it is ever read, and that this write occurs before any read of the location. Any location whose value can be written after initialization is *mutable*. Any location whose value cannot be written after initialization is *immutable*. Mutable locations include mutable variables and `settable` fields of a reference object. Immutable locations include non-`transient`, non-`settable` fields of a reference object.

4.4 Threads and Parallelism

There are two kinds of threads in Fortress: *implicit* threads and *spawned* (or *explicit*) threads. Spawned threads are objects created by the `spawn` construct, described in Section 13.23.

A thread may be in one of five states: *not started*, *executing*, *suspended*, *normally completed* or *abruptly completed*. We say a thread is *not started* after it is created but before it has taken a step. This is only important for purposes of determining whether two threads are executing simultaneously; see Section 32.4. A thread is *executing* or *suspended* if it has taken a step, but has not completed; see below for the distinction between executing and suspended threads. A thread is *complete* if its expression has completed evaluation. If the expression completes normally, its value is the result of the thread.

Every thread has a *body* and an *execution environment*. The body is an intermediate expression, which the thread evaluates in the context of the execution environment; both the body and the execution environment may change when the thread takes a step. This environment is used to look up names in scope but bound in a block that encloses the construct that created the thread. The execution environment of a newly created thread is the environment of the thread that created the new thread.

In Fortress, a number of constructs are *implicitly parallel*. An implicitly parallel construct creates a *group* of one or more implicit threads. The implicitly parallel constructs are:

- **Tuple expressions:** Each element expression of the tuple expression is evaluated in a separate implicit thread (see Section 13.27).
- **also do blocks:** each sub-block separated by an `also` clause is evaluated in a separate implicit thread (see Section 13.11.3).
- **Method invocations and function calls:** The receiver or the function and each of the arguments is evaluated in a separate implicit thread (see Section 13.4, Section 13.5, and Section 13.6).
- **for loops, comprehensions, sums, generated expressions, and big operators:** Parallelism in looping constructs is specified by the generators used (see Section 13.14). Programmers should assume that generators other than the *sequential* generator can execute each iteration of the body expression in a separate implicit thread. Reduction variables in `for` loops, and results returned from comprehensions and big operators correspond to reductions (see Section 4.4.1).
- **Extremum expressions:** Each guarding expression of the extremum expression is evaluated in a separate implicit thread (see Section 13.20).
- **Tests:** Each test is evaluated in a separate implicit thread (see Chapter 19).

Implicit threads run fork-join style: all threads in a group are created together, and they all must complete before the group as a whole completes. There is no way for a programmer to single out an implicit thread and operate upon it in any way; they are managed purely by the compiler, runtime, and libraries of the Fortress implementation. Implicit threads need not be scheduled fairly; indeed, a Fortress implementation may choose to serialize portions of any group of implicit threads, interleaving their operations in any way it likes. For example, the following code fragment may loop forever:

```
r : ℤ64 := 0
(r := 1, while r = 0 do end)
```

If any implicit thread in a group completes abruptly, the group as a whole will complete abruptly as well. Every thread in the group will either not run at all, complete (normally or abruptly), or may be terminated early as described in Section 32.6. The result of the abrupt completion of the group is the result of one of the constituent threads that completes abruptly. After abrupt completion of a group of implicit threads, each reduction variable (see Section 4.4.1) may reflect an arbitrary subset of updates performed by the threads in the group. This means in general that reduction

variables should not be accessed after abrupt completion. The exact behavior of reduction variables depends on the supporting generator structure and is described in Section 32.8.

Spawned thread objects are reference objects of type `Thread[[T]]`, where T is the static type of the expression spawned; this trait has the following methods, each taking no arguments:

- The *val* method returns the value computed by the subexpression of the `spawn` expression. If the thread has not yet completed execution, the invocation of *val* blocks until it has done so. A spawned thread is not permitted to exit (discussed in Section 13.12) to a surrounding label (the label is considered to be out of scope), but may fail to catch an exception and thus complete abruptly. When this happens, the exception is *deferred*. Any invocation of the *val* method on the spawned thread throws the deferred exception. If the spawned thread object is discarded, the exception will be silently ignored. In particular, if the result of a `spawn` expression is dropped, it is impossible to detect completion of the spawned thread or to recover from any deferred exceptions.
- The *wait* method waits for a thread to complete, but does not return a value and will not throw a deferred exception.
- The *ready* method returns *true* if a thread has completed, and returns *false* otherwise.
- The *stop* method attempts to terminate a thread as described in Section 32.6.

We say a spawned thread has been *observed to complete* after invoking the *val* or *wait* methods, or when an invocation of the *ready* method returns *true*.

In the absence of sufficient parallel resources, an attempt is made to run the subexpression of the `spawn` expression before continuing succeeding evaluation. We can imagine that it is actually the *rest* of the evaluation *after* the parallel block which is spawned off in parallel. This is a subtle technical point, but makes the sequential execution of parallel code simpler to understand, and avoids subtle problems with the asymptotic stack usage of parallel code [21, 11].

There are three ways in which a thread can be suspended. First, a thread that begins evaluating an implicitly parallel construct, creating a thread group, is suspended until that thread group has completed. Second, a thread that invokes *val* or *wait* is suspended until the spawned thread completes. Finally, invoking the *abort* function from within an `atomic` expression may cause a thread to suspend; see Section 32.3.

Threads in a Fortress program can perform operations simultaneously on shared objects. In order to synchronize data accesses, Fortress provides `atomic` expressions (see Section 13.22). Chapter 21 describes the memory model which is obeyed by Fortress programs.

4.4.1 Reduction Variables

To perform computations as locally as possible, and avoid the need to synchronize in the middle of relatively simple for loops, Fortress gives special treatment to *reductions*. We say that an operator \odot is a *reduction operator* for type T if T is a subtype of `Monoid[[T, \odot]]`, which implies that \odot is an associative binary infix operator on T (see Section 37.5 for details about the `Monoid` trait).

We say that a variable l is a *reduction variable* reduced using the reduction operator \odot for a particular thread group if it satisfies the following conditions:

- Every assignment to l within the thread group is of the form $l \oplus = e$, where exactly one operator \oplus or its group inverse \ominus (see below) is used in these assignments.
- The value of l is not otherwise read within the thread group.
- The variable l is not a free variable of a functional. This includes the fields of the receiver object in a method definition.

Other threads which simultaneously reference a reduction variable while a loop is running may see an arbitrary value for that variable. Any updates performed by those threads may be lost. The association of terms in the reduction is arbitrary and guided by the loop generators (Section 32.8). The order of terms in the reduction is the natural order of elements in the generator. When the operator is commutative, the order of terms is also arbitrary.

Several common mathematical operators are declared to be monoids in the Fortress standard libraries. These include $+$, \cdot , \wedge , \vee , and $\underline{\vee}$. If a type T extends `Group[[T, ⊕, ⊖]]` (see Section 37.5 for details about the `Group` trait) then reduction expressions of the following form are also permitted:

$$x \ominus= y$$

Such expressions may be transformed into an algebraic equivalent such as the following:

$$x \oplus= \text{Identity}[\oplus] \ominus y$$

The semantics of reductions enables implementation strategies such as the one used in OpenMP [24]: A reduction variable l is assigned `Identity[[⊕]]` at the beginning of each iteration. The original variable value may be read ahead of time, resulting in the loss of parallel updates to the variable which occur in other threads while the loop is running. When all iterations are complete, the initial value of the reduction variable and values of the variable at the end of each implicit thread are reduced and the result is assigned to the reduction variable as the group completes.

In the following example, `sum` is a reduction variable:

```
arraySum[[nat x]](a : Z64[x]) : Z64 = do
  sum : Z64 := 0
  for i ← a.indices do
    sum += a[i]
  end
  sum
end
```

The following demonstrates the use of a reduction variable `prod` in a parallel `do`:

```
prod : Z64 := 1
do
  prod := g(y)
also do
  prod := f(x)
  prod := g(x)
also do
  h(x, y)
end
```

The operator associated with a reduction variable can be invoked in several places during evaluation of a thread group. Most obviously, it can be invoked anywhere the reduction operation occurs textually in the code for that group. The operator can also be invoked multiple times implicitly and in parallel after the completion of some or all of the threads in the group. The group as a whole does not complete normally until all these operator invocations have completed normally and the reduced result has been assigned to the reduction variable for use in subsequent computation. If any operator invocation completes abruptly, it is treated just as if an ordinary implicit thread in the group completed abruptly.

Different implicit threads (even different iterations of the same loop body) may execute in very different amounts of time. A naively parallelized loop will cause processors to idle until every iteration finishes. The simplest way to mitigate this delay is to expose substantially more parallel work than the number of underlying processors available to run them. Load balancing can move the resulting (smaller) units of work onto idle processors to balance load.

The ratio between available work and number of threads is called *parallel slack* [3, 4]. With support for very lightweight threading and load balancing, slack in hundreds or thousands (or more) proves beneficial; very slack

computations easily adapt to differences in the number of available processors. Slack is a desirable property, and the Fortress programmer should endeavor to expose parallelism where possible.

4.5 Environments

An *environment* maps *names* to values or locations. Environments are immutable, and two environments that map exactly the same names to the same values or locations are identical.

A program starts executing with an empty environment. Environments are extended with new mappings by variable, function, and object declarations, and functional calls (including calls to object constructors). After initializing all top-level variables and singleton objects as described in Section 22.6, the *top-level environment* for each component is constructed.

The environment of a value is determined by how it is *constructed*. For all but object expressions, function expressions and local function declarations, the environment of the constructed value is the top-level environment of the component in which the expression or declaration occurs. For object and function expressions and local function declarations, the environment of the constructed value is the lexical environment in which the expression or declaration was evaluated.

We carefully distinguish a spawned thread from its associated spawned thread object. In particular, note that the execution environment of a spawned thread, in which the body expression is evaluated, is distinct from the environment of the associated thread object, in which calls to the thread methods are evaluated.

4.6 Input and Output Actions

Certain functionals in Fortress perform primitive input/output (I/O) actions. These actions have an externally visible effect. Any functional which may perform an I/O action—either because it is a primitive action or because it invokes other functionals which perform I/O actions—must be declared with the `io` modifier.

Any primitive I/O action may take many internal steps; each step may read or write any memory locations referred to either directly or transitively by object references passed as arguments to the action. Each I/O action is free to complete either normally or abruptly. I/O actions may block and be prevented from taking a step until any necessary external conditions are fulfilled (input is available, data has been written to disk, and so forth).

Each I/O action taken by an expression is considered part of that expression's effects. The steps taken by an I/O action are considered part of the context in which an expression evaluates, in much the same way as effects of simultaneously-executing threads must be considered when describing the behavior of an expression. For example, we cannot consider two functionals to be equivalent unless the possible I/O actions they take are the same, given identical internal steps by each I/O action.

Chapter 5

Lexical Structure

A Fortress program consists of a finite sequence of Unicode 5.0 abstract characters [27]. Every character in a program is part of an input element. The partitioning of the character sequence into input elements is uniquely determined by the characters themselves. In this chapter, we explain how the sequence of input elements of a program is determined from a program’s character sequence.

This chapter also describes standard ways to *render* (that is, *display*) individual input elements in order to approximate conventional mathematical notation more closely. Other rules, presented in later chapters, govern the rendering of certain *sequences* of input elements; for example, the sequence of three input elements a , \wedge , and b may be rendered a^b . The rules of rendering are “merely” a convenience intended to make programs more readable. Alternatively, the reader may prefer to think of the rendered presentation of a program as its “true form” and to think of the underlying sequence of Unicode characters as “merely” a convenient way of encoding mathematical notation for keyboarding purposes.

Most of the program text in this specification is shown in rendered presentation form. However, sometimes, particularly in this chapter, unformatted code is presented to aid in exposition. In many cases the unformatted form is shown alongside the rendered form in a table, or following the rendered form in parentheses; as an example, consider the operator \oplus (OPLUS).

5.1 Characters

A Unicode 5.0 abstract character is the smallest element of a Fortress program.¹ Many characters have standard *glyphs*, which are how these characters are most commonly depicted. However, more than one character may be represented by the same glyph. Thus, Unicode 5.0 specifies a representation for each character as a sequence of *code points*. Each character used in this specification maps to a single code point, designated by a hexadecimal numeral preceded by “U+”. Unicode also specifies a name for each character;² when introducing a character, we specify its code point and name, and sometimes the glyph we use to represent it in this specification. In some cases, we use such glyphs without explicitly introducing the characters (as, for example, with the simple upper- and lowercase letters of the Latin and Greek alphabets). When the character represented by a glyph is unclear and the distinction is important, we specify the code point or the name (or both). The Unicode Standard [27] specifies a *general category* for each character, which we use to describe sets of characters below.

¹Note that a single Unicode abstract character may have multiple encodings. Fortress does not distinguish between different encodings of the same character that may be used in representing source code, and it treats canonically equivalent characters as identical. See the Unicode Standard [27] for a full discussion of encoding and canonical equivalence.

²There are sixty-five “control characters”, which do not have proper names. However, many of them have *Unicode 1.0 names*, or other standard names specified in the Unicode Character Database, which we use instead.

We partition the Unicode 5.0 character set into the following (disjoint) classes:

- *special non-operator characters*, which are:

U+0026	AMPERSAND	&	U+0027	APOSTROPHE	'
U+0028	LEFT PARENTHESIS	(U+0029	RIGHT PARENTHESIS)
U+002C	COMMA	,	U+002E	FULL STOP	.
U+003B	SEMICOLON	;	U+005C	REVERSE SOLIDUS	\
U+2026	HORIZONTAL ELLIPSIS	...	U+21A6	RIGHTWARDS ARROW FROM BAR	↩
U+2200	FOR ALL	∀	U+2203	THERE EXISTS	∃
U+2254	COLON EQUALS	:=			
U+27E6	MATHEMATICAL LEFT WHITE SQUARE BRACKET	[
U+27E7	MATHEMATICAL RIGHT WHITE SQUARE BRACKET]			

- *special operator characters*, which are

U+003A	COLON	:	U+003D	EQUALS SIGN	=
U+005B	LEFT SQUARE BRACKET	[U+005D	RIGHT SQUARE BRACKET]
U+005E	CIRCUMFLEX ACCENT	^	U+007B	LEFT CURLY BRACKET	{
U+007C	VERTICAL LINE		U+007D	RIGHT CURLY BRACKET	}
U+2192	RIGHTWARDS ARROW	→	U+21D2	RIGHTWARDS DOUBLE ARROW	⇒

- *letters*, which are characters with Unicode general category Lu, Ll, Lt, Lm or Lo—those with Unicode general category Lu are *uppercase letters*—and the following *honorary letters*:

U+221E	INFINITY	∞	U+22A4	DOWN TACK	‡	U+22A5	UP TACK	⌞
--------	----------	---	--------	-----------	---	--------	---------	---

- *connecting punctuation* (Unicode general category Pc);

- *digits* (Unicode general category Nd);

- *prime characters*, which are:

U+2032	PRIME	'	U+2033	DOUBLE PRIME	''
U+2034	TRIPLE PRIME	'''	U+2035	REVERSED PRIME	ƶ
U+2036	REVERSED DOUBLE PRIME	Ʒ	U+2037	REVERSED TRIPLE PRIME	Ƹ

- *whitespace characters*, which are *spaces* (Unicode general category Zs) and the following characters:

U+0009	CHARACTER TABULATION		U+000A	LINE FEED	
U+000B	LINE TABULATION		U+000C	FORM FEED	
U+000D	CARRIAGE RETURN				
U+001C	INFORMATION SEPARATOR FOUR		U+001D	INFORMATION SEPARATOR THREE	
U+001E	INFORMATION SEPARATOR TWO		U+001F	INFORMATION SEPARATOR ONE	
U+2028	LINE SEPARATOR		U+2029	PARAGRAPH SEPARATOR	

- *character literal delimiters*, which are:

U+0060	GRAVE ACCENT	`
U+2018	LEFT SINGLE QUOTATION MARK	‘
U+2019	RIGHT SINGLE QUOTATION MARK	’

- *string literal delimiters*, which are:

U+0022	QUOTATION MARK	"
U+201C	LEFT DOUBLE QUOTATION MARK	“
U+201D	RIGHT DOUBLE QUOTATION MARK	”

- *ordinary operator characters*, enumerated (along with the special operator characters) in Appendix F, which include the following characters with code points less than U+007F:

U+0021	EXCLAMATION MARK	!	U+0023	NUMBER SIGN	#
U+0024	DOLLAR SIGN	\$	U+0025	PERCENT SIGN	%
U+002B	PLUS SIGN	+	U+002D	HYPHEN-MINUS	-
U+003F	QUESTION MARK	?	U+0040	COMMERCIAL AT	@
U+002A	ASTERISK	*	U+002F	SOLIDUS	/
U+003C	LESS-THAN SIGN	<	U+003E	GREATER-THAN SIGN	>
U+007E	TILDE	~			

and most (but not all) Unicode characters specified to be *mathematical operators* (i.e., characters with code points in the range 2200-22FF) are operators in Fortress.

- *other characters*

Some other classes of characters, which overlap with the ones above, are useful to distinguish:

- *control characters* are those with Unicode general category Cc;
- *ASCII characters* are those with code points U+007F and below;
- *protected characters* are the ASCII characters, control characters, and string literal delimiters;
- *word characters* are letters, digits, connecting punctuation, prime characters, and apostrophe;
- *restricted-word characters* are ASCII letters, ASCII digits, and the underscore character (i.e., ASCII word characters other than apostrophe);
- *hexadecimal digits* are the digits and the ASCII letters A, B, C, D, E, F, a, b, c, d, e and f;
- the *digit-group separator* is NARROW NO-BREAK SPACE (U+202F);
- *operator characters* are special operator characters and ordinary operator characters;
- *special characters* are special non-operator characters and special operator characters;
- *enclosing characters* are the enclosing operator characters enumerated in Section F.1, left and right parenthesis characters, and mathematical left and right white square brackets;

Forbidden and Restricted Characters

It is a static error for a Fortress program to contain any control character other than the above-listed whitespace characters, except that the character SUBSTITUTE (U+001A; also known as “control-Z”) is allowed and ignored if it is the last character of a program.

It is a static error for the following characters to occur outside a comment:

U+0009	CHARACTER TABULATION	U+000B	LINE TABULATION
U+001C	INFORMATION SEPARATOR FOUR	U+001D	INFORMATION SEPARATOR THREE
U+001E	INFORMATION SEPARATOR TWO	U+001F	INFORMATION SEPARATOR ONE

Thus, LINE FEED, FORM FEED and CARRIAGE RETURN are the only control characters—and the only whitespace characters other than spaces, LINE SEPARATOR, and PARAGRAPH SEPARATOR—outside of comments in a valid Fortress program.

It is a static error for REVERSE SOLIDUS (‘\’;U+005C), any connecting punctuation other than LOW LINE (‘_’; U+005F, also known as SPACING UNDERSCORE), any character in the “other characters” class above, or the following whitespace characters to appear outside of comments and string and character literals:

U+2000	EN QUAD	U+2003	EM SPACE	U+2006	SIX-PER-EM SPACE
U+2001	EM QUAD	U+2004	THREE-PER-EM SPACE	U+2007	FIGURE SPACE
U+2002	EN SPACE	U+2005	FOUR-PER-EM SPACE	U+2008	PUNCTUATION SPACE

5.2 Words and Chunks

A *chunk* of a Fortress program is a nonempty contiguous subsequence of the program.

A *word* of a Fortress program is a maximal chunk consisting of only word characters (letters, digits, connecting punctuation, prime characters and apostrophe); that is, a word is one or more consecutive word characters delimited by characters other than word characters (or the beginning or end of the program).

A *restricted word* is a maximal chunk that has only restricted-word characters (ASCII letters and digits and underscore characters). Note that a restricted word is *not* a word if it is delimited by word characters.

5.3 Lines, Pages and Position

The characters in a Fortress program are partitioned into *lines* and *pages*, delimited by *line terminators* and *page terminators* respectively. A *page terminator* is an occurrence of the character `FORM_FEED`. A *line terminator* is an occurrence of any of the following:

- `LINE FEED`,
- `CARRIAGE RETURN` not immediately followed by `LINE FEED`,
- `LINE SEPARATOR`, or
- `PARAGRAPH SEPARATOR`.

A character is on page n (respectively line n) of a program if exactly $n - 1$ page terminators (respectively line terminators) precede that character in the program. Thus, for example, the n th line terminator of a program is the last character on line n . A character is on line k of page n if it is on page n and is preceded by exactly $k - 1$ line terminators on page n . A character is at line position k on line n if it is preceded by exactly $k - 1$ characters on line n other than page terminators. Note that a page terminator does *not* terminate a line; thus, the character immediately following a page terminator need not be at line position 1.

Before any other processing, a Fortress program undergoes *ASCII conversion* (see Section 5.4), which may replace chunks of ASCII characters with single (typically non-ASCII) characters and remove some other characters. We expect that IDEs will typically display a program by rendering the converted sequence of characters rather than the actual input sequence. Thus, a program may appear to have fewer (and different) characters than it actually does. Nonetheless, the page, line and position of a character is based on the program before conversion.³

If a character (or any other syntactic entity) x precedes another character y in the program, we say that x is *to the left of* y and that y is *to the right of* x , regardless of how they may appear in a typical rendered display of the program. Thus, it is always meaningful to speak, for example, of the left-hand and right-hand operands of a binary operator, or the left-hand side of an assignment expression.

5.4 ASCII Conversion

To facilitate interaction with legacy tools, and particularly to aid in program entry, Fortress provides an ASCII encoding for programs. To support this encoding, a Fortress program undergoes *ASCII conversion*, which is described in detail in Appendix E. This section gives a high-level overview of ASCII conversion.

³Of course, an IDE may instead do the conversion and manipulate the converted program, in which case, the page, line and position of characters will reflect the conversion.

ASCII conversion takes a program and yields an equivalent program. For every valid Fortress program, there is an equivalent program that contains only ASCII characters.⁴ ASCII conversion is idempotent: converting a program resulting from ASCII conversion yields the same program. Unless otherwise specified, all constraints and properties of Fortress programs stipulated in this specification apply to the programs after ASCII conversion.

ASCII conversion consists of three steps. The first step consists of “pasting” words across line breaks, so that long identifiers and numerals can be split across lines. Identifiers may be very long because non-ASCII Unicode characters may be encoded by long chunks of ASCII characters (the actual conversion of such chunks is done in the next step). Roughly speaking, two consecutive lines are pasted together if the first ends with an ampersand that is immediately preceded by a word character, and the second begins with an ampersand that is immediately followed by a word character.

The second step replaces certain restricted words, and sequences of operator and special characters, with single Unicode characters. Roughly speaking, if a restricted word is either the official Unicode 5.0 name with underscores in place of spaces and hyphens, or a specified alternative name, of some character that is not a protected character (ASCII and control characters and string literal delimiters), then the restricted word is replaced by that character. In some cases, even a fragment of a restricted word may be replaced by a single character (most commonly a Greek letter). Some multicharacter sequences of ASCII operator and special characters are also replaced by non-ASCII operator or special characters; we call such sequences *ASCII shorthand*. However, within string literals, neither restricted words nor ASCII shorthand is converted. Instead, we provide *escape sequences* to get non-ASCII characters within strings (see Section 5.10).

The third step replaces apostrophes that appear within what would otherwise be numerals with digit-group separators.

Precise descriptions of these steps are given in Appendix E, including the rules for replacing fragments of restricted words and the specification of alternative names and ASCII shorthand for non-operator characters. Alternative names and ASCII shorthand for operator characters are given in Appendix F.

The ampersand is particularly important in the ASCII conversion process, and to preserve the idempotence of ASCII conversion, its use is severely restricted in the converted program. In particular, outside of comments and string and character literals, its only legitimate use is in “cancelling” line breaks, as described in the Section 5.7. This restriction is also stated precisely in Appendix E.

5.5 Input Elements and Scanning

After ASCII conversion, a Fortress program is partitioned into *input elements* by a process called *scanning*.⁵ Scanning transforms a Fortress program from a sequence of Unicode characters to a sequence of input elements. Input elements are always chunks: the characters that comprise an input element always appear contiguously in the program. Every input element is a *whitespace element* (*comments* are included in whitespace elements) or a *token*. Every token is a *reserved word*, a *literal*, an *identifier*, an *operator token*, or a *special token*. There are five kinds of literals: boolean literals, character literals, string literals, the void literal, and numerals (i.e., numeric literals).

Conceptually, we can think of scanning as follows: First, the comments, character literals and string literals are identified. Then the remaining characters are divided into words (i.e., chunks of letters, digits, connecting punctuation, primes and apostrophes), whitespace characters, and other characters. In some cases, words separated by a single ‘.’ or digit-group separator (and no other characters) are joined to form a single numeral (see Section 5.13). Words that are not so joined are classified as reserved words, boolean literals, numerals, identifiers, or operator tokens, as described in later sections in this chapter. It is a static error if any word in a Fortress program cannot be classified as one of these. All remaining whitespace characters, together with the comments, form whitespace elements, which may be *line-breaking*. Finally, chunks of symbols (and a few other special cases) are checked to see whether they

⁴See Appendix E for the precise notion of equivalence guaranteed by ASCII conversion.

⁵Fortress has a facility for defining new syntax, discussed in Chapter 36. However, except for that chapter, this specification describes the Fortress language only for programs that use the standard Fortress syntax.

form multicharacter operator tokens, as described in Section 5.14, or the void literal (see Section 5.12). Every other character is a token by itself, either a special token (if it is a special character) or an operator token.

5.6 Comments

The character sequences “`(*`” and “`*)`” are referred to as the *opening comment delimiter* and the *closing comment delimiter* respectively. In a valid program, every occurrence of the opening comment delimiter (outside of string literals) is balanced by a subsequent occurrence of the closing comment delimiter; it is a static error if comment delimiters are not properly balanced. All the characters between a balanced pair of comment delimiters, including the comment delimiters themselves, comprise a *comment*. Comments may be nested.

5.7 Whitespace Elements

A whitespace element is a maximal chunk consisting of comments, whitespace characters that are not within string or character literals or numerals, and ampersands that are not within string or character literals.

We distinguish *line-breaking whitespace* from *non-line-breaking whitespace* using the following terminology:

- A *line-terminating comment* is a comment that encloses one or more line terminators. All other comments are called *spacing comments*.
- *Spacing* refers to any chunk of spaces, `FORM FEED` characters and spacing comments.
- A *line break* is a line terminator or line-terminating comment that is not immediately preceded by an ampersand (U+0026), possibly with intervening spacing.
- *Whitespace* refers to any nonempty sequence of spacing, ampersands, line terminators, and line-terminating comments.
- *Line-breaking whitespace* is whitespace that contains at least one line break.

It is a static error if an ampersand occurs in a program (after ASCII conversion) unless it is within a character or string literal or a comment, or it is immediately followed by a line terminator or line-terminating comment (possibly with intervening spacing).

5.8 Reserved Words

The following tokens are *reserved words*:

BIG	SI_unit	absorbs	abstract	also	api	as
asif	at	atomic	bool	case	catch	coerces
coercion	component	comprises	default	dim	do	elif
else	end	ensures	except	excludes	exit	export
extends	finally	fn	for	forbid	from	getter
hidden	ident	if	import	in	int	invariant
io	juxtaposition	label	largest	nat	object	of
opr	or	private	property	provided	requires	self
settable	setter	smallest	spawn	syntax	test	then
throw	throws	trait	transient	try	tryatomic	type
typecase	unit	value	var	where	while	widening
widens	with	wrapped				

The operators on units, namely cubed, cubic, in, inverse, per, square, and squared, are also reserved words.

To avoid confusion, Fortress reserves the following tokens:

```
goto  idiom  public  pure  reciprocal  static
```

They do not have any special meanings but they cannot be used as identifiers.

5.9 Character Literals

A character literal consists of one or more characters enclosed in single quotation marks. The character that begins a character literal is called the literal's *opening mark*; the character that ends it is its *closing mark*. For convenience, the marks may be true typographical “curly” single quotation marks (U+2018 and U+2019), a pair of apostrophe characters (U+0027), or a “backquote” character (U+0060) and an apostrophe character. It is a static error if the opening and closing marks do not match: a character literal beginning with a left single quotation mark must end with a right single quotation mark; one beginning with either an apostrophe or a backquote must end with an apostrophe. As discussed in Section 13.1, a character literal evaluates to a value of type `Character` (see Section 8.7), which represents an abstract Unicode character.

A left single quotation mark (U+2018) or a backquote begins a character literal unless it is within a comment, another character literal, or a string literal. An apostrophe (U+0027) begins a character literal unless it is within a comment, another character literal, or a string literal, or it is immediately preceded by a word character (i.e., a letter, digit, connecting punctuation, prime, or apostrophe). In either case, the character literal ends with the nearest apostrophe or right single quotation mark *after* the first character following the opening mark. In particular, an apostrophe or right single quotation mark immediately following the opening mark of a character literal is *not* a closing mark. Thus, for example, the sequence `' '` is a single character literal with one enclosed character `'`.

It is a static error if any of the enclosed characters of a character literal is a line feed, form feed, carriage return, or any character forbidden outside comments in a Fortress program (see Section 5.1), or if there is exactly one enclosed character and it is a backslash (U+005C). It is also a static error if any of the enclosed characters is a string literal delimiter that is not immediately preceded by a backslash (i.e., an *unescaped* string literal delimiter). This last restriction is necessary to prevent ASCII conversion from changing the boundaries of string literals (see Appendix E).

The sequence of enclosed characters may be a single character (e.g., `'a'`, `'$'`, `'α'`, `'⊕'`), a sequence of four or more hexadecimal digits identifying the code point of a Unicode character (e.g., `'001C'`, `'FBAB'`, `'1D11E'`), the official Unicode 5.0 name or an alternative name of a Unicode character with spaces and hyphens intact (e.g., `'PLUS-MINUS SIGN'`), a sequence of ASCII characters that would be converted by the second step of ASCII conversion to a single Unicode character (see Section E.2), or a *character-literal escape sequence*. The character-literal escape sequences, and the characters such character literals evaluate to, are:

```
\b  U+0008  BACKSPACE
\t  U+0009  CHARACTER TABULATION
\n  U+000A  LINE FEED
\f  U+000C  FORM FEED
\r  U+000D  CARRIAGE RETURN
\"  U+0022  QUOTATION MARK
\\  U+005C  REVERSE SOLIDUS
\“  U+201C  LEFT DOUBLE QUOTATION MARK
\”  U+201D  RIGHT DOUBLE QUOTATION MARK
```

It is a static error if the sequence of enclosed characters is not one of the kinds listed above. In particular, it is a static error if the hexadecimal digits enclosed do not correspond to the code point of a Unicode 5.0 abstract character.

Note that ASCII conversion is performed within character literals. Thus a character literal written as

'GREEK_CAPITAL_LETTER_LAMBDA'

is equivalent to a character literal written as

'Λ'

Although names for control and other protected characters are *not* converted during ASCII conversion, they *are* permitted within character literals. Both the standard form of such names (i.e., with spaces and hyphens) and the form with spaces and hyphens replaced by underscore characters are permitted. Thus, for example, the literals 'BACKSPACE', '_ and 'U+0008' all evaluate to the BACKSPACE character.

5.10 String Literals

A string literal is a sequence of characters enclosed in double quotation marks (for example, "Hello, world!" or " πr^2 "). The character that begins a string literal is the literal's *opening mark*; the character that ends it is its *closing mark*. For convenience, the opening and closing marks of a string literal may be either true typographical “curly” double quotation marks (U+201C and U+201D) or a pair of “neutral” double-quote characters. It is a static error if the opening and closing marks of a string literal do not “match”, that is, if one is “curly” and the other “neutral”. As discussed in Section 13.1, a string literal evaluates to a value of type String (see Section 8.7), which represents a finite sequence of abstract Unicode characters.

A left double quotation mark (U+201C) or “neutral” quotation mark (U+0022) begins a string literal unless it is within a comment, a character literal, or another string literal. It ends with the nearest following unescaped (i.e., not immediately preceded by an unescaped backslash) right double quotation mark or “neutral” quotation mark. Therefore, it is not possible for a string literal to include an unescaped right double quotation mark or “neutral” quotation mark as an enclosed character. In addition, it is a static error for an unescaped left double quotation mark to be an enclosed character of a string literal.

Within a string literal, a backslash introduces an *escape sequence*, unless it is immediately preceded by an odd number of backslashes, in which case the backslash is itself escaped. There are three kinds of escape sequences recognized within a string literal: the character-literal escape sequences (see Section 5.9), *restricted-word escape sequences*, and *quoted-character escape sequences*.

A restricted-word escape sequence consists of an unescaped backslash immediately followed by a restricted word not beginning with a lowercase letter. It is a static error for a string literal to contain any restricted-word escape sequence other than “\BACKSPACE”, “\TAB”, “\NEWLINE”, “\FORMFEED”, or “\RETURN”.

A quoted-character escape sequence consists of an unescaped backslash immediately followed by an apostrophe and the sequence of characters immediately following the apostrophe up to and including the next apostrophe. Note that this kind of escape sequence looks just like a backslash followed by a character literal. It is a static error if such a sequence with the initial backslash removed would not be a valid character literal (see Section 5.9). Quoted-character escape sequences are useful for ASCII “shorthands” that begin with a lowercase letter or are not restricted words, and when the escape sequence is immediately followed in the string by a letter, digit, or underscore. For example, "\beta" evaluates to a string containing a backspace character (indicated by \b) followed by the letters e, t, and a, but "\'beta'" evaluates to a string containing the single letter β . As another example, "x\AND\'NOT'y" becomes "x \wedge y", but "x\AND\NOTy" is a static error, because the name “NOTy” does not correspond to a Unicode character. Also, the string "F\o\ [\ ' T \ ' \]'" becomes "F\o\ [T]".

It is a static error if an unescaped backslash is immediately followed by a character other than another backslash, an apostrophe, a string literal delimiter, or a restricted-word character. In addition, it is also a static error if an unescaped backslash within a string literal is followed by any lowercase letter other than ‘b’, ‘t’, ‘n’, ‘f’ or ‘r’. This rule preserves a certain level of compatibility with the C and Java programming languages.

Unlike elsewhere in a program, the enclosed characters of a string literal are *not* subject to the second or third step of ASCII conversion (word pasting across line terminators still occurs within string literals). Also, the formatting of identifiers and numerals described in Section 5.17 is not performed within string literals.

5.11 Boolean Literals

The boolean literals are *false* and *true*.

5.12 The Void Literal

The void literal is `()` (pronounced “void”).

5.13 Numerals

A numeric literal, or *numeral*, in Fortress is a maximal chunk consisting one or more words (and the intervening characters) satisfying the following properties:

- each word consists of only digits and letters, except that the last word may have one underscore as part of a *radix specifier*, defined below;
- consecutive words are separated by exactly one character, either a digit-group separator or a ‘.’ character; and
- either the first word begins with a digit or the last word has a radix specifier.

A *radix specifier* is a word suffix consisting of an underscore and then either by a sequence of one or more digits or by the English name in all uppercase ASCII letters of an integer from 2 to 16. The *radix* of a numeral with a radix specifier is the number denoted by the sequence of digits (interpreted in base 10) or named by the ASCII letters of the radix specifier. It is a static error if the radix of a numeral is not an integer from 2 to 16. It is also a static error if any of the following conditions hold:

- the numeral contains letters and does not have a radix specifier;
- the numeral contains letters other than A, B, C, D, E, F, a, b, c, d, e or f, and has a radix other than 12;
- the numeral has radix 12 and contains letters other than A, B, X, E a, b, x or e, or contains at least one A, B, a or b and at least one X, E, x or e;
- the numeral has a radix specifier and contains a digit or letter that denotes a value greater than or equal to the numeral’s radix;⁶ or
- the numeral contains both uppercase and lowercase letters.

A numeral is *simple* if it does not contain a ‘.’ character; otherwise, the number is *compound*. It is a static error if a numeral contains more than one ‘.’ character.

Here are some examples of valid numerals, shown in unformatted form:

```
17 7fff_16 0fff_SIXTEEN 10101101_2 XE_12 3.14159265 DEAD.BEEF_16
```

On the other hand, the following would be classified as numerals but then rejected as static errors:

```
0.a 0x6A35 FF_EIGHT 12.52.23 57_50 PI_FIFTEEN dead.BEEF_16
```

⁶The value denoted by a digit is specified by the Unicode Standard [27]. The value denoted by the letters are as follows: A or a: 10; B or b: 11; C or c: 12; D or d: 13; E or e: 11 if the radix is 12, 14 otherwise; F or f: 15; X or x: 10.

5.14 Operator Tokens

In this section, we describe how to determine the operator tokens of a program. Because operator tokens do not occur within comments, string literals and character literals, we henceforth in this section consider only characters that are outside these constructs.

An *operator word* of a program is a word that is not reserved, consists only of uppercase letters and underscores (no digits or non-uppercase letters), does not begin or end with an underscore, and has at least two different letters.

A *base operator* is an ordinary operator character,⁷ the two-character sequence “**”, an operator word, a (contiguous) sequence of two or more vertical-line characters (U+007C), or a multicharacter enclosing operator, as defined in Section 5.14.1. A base operator is *maximal* of a program if it is not contained within any other base operator of that program. It is a static error if two maximal base operators overlap (which is only possible if both are multicharacter enclosing operators). A *simple operator* is a maximal base operator that is an operator character, the two-character sequence “**”, or an operator word.

A maximal base operator is an operator token unless it is a simple operator other than an enclosing or vertical-line operator character, and it is immediately preceded by “^” or immediately followed by “=”. Such an operator is an *enclosing operator* if it is an enclosing operator character (see Section F.1) or a multicharacter enclosing operator; it is a *vertical-line operator* if it has only vertical-line operator characters (see Section F.2); otherwise, it is an *ordinary operator*.

If a simple operator is immediately preceded by “^” then the “^” followed by the simple operator is an operator token; such an operator token is called a *superscripted postfix operator*. In addition, “^T” is also a superscripted postfix operator provided that it is not immediately followed by a word character. It is a static error for a superscripted postfix operator to be immediately followed by a word character other than apostrophe.

Finally, if a simple operator is not immediately preceded by “^” and is immediately followed by “=” then the simple operator followed by the “=” is an operator token; such an operator token is called a *compound assignment operator*.

5.14.1 Multicharacter Enclosing Operators

The following multicharacter sequences (in which there must be no other characters, and particularly no whitespace) can be used as brackets as described below:

1. Any chunk of vertical-line characters is a vertical-line operator. Such an operator can be used in an enclosing pair matching itself.
2. Any of ‘(’ or ‘[’ or ‘{’ may be immediately followed by any number of ‘/’ characters or by any number of ‘\’ characters. Such a token is a left bracket, and it is matched by the multicharacter token consisting of the same number and kind of ‘/’ or ‘\’ characters followed immediately by a matching ‘)’ or ‘]’ or ‘}’, as appropriate. Thus, for example, “(/////” and “/////)” are matching left and right brackets respectively. (In the future, we may allow tokens with mixtures of ‘/’ and ‘\’, in which case the left and right brackets should match from outside in. But for now, such tokens are simply illegal.)
3. One or more ‘<’ characters may be followed immediately by one or more ‘|’ characters. Such a token is a left bracket, and it is matched by the multicharacter token consisting of the same number of ‘|’ characters followed by as many ‘>’ characters as there are ‘<’ characters in the left bracket.
4. One or more ‘<’ characters, or one or more ‘|’ characters may be followed immediately by one or more ‘/’ characters or by one or more ‘\’ characters. Such a token is a left bracket, and it is matched by the multicharacter token consisting of the same number and *opposite* kind of ‘/’ or ‘\’ characters followed immediately by as

⁷The operator characters are enumerated in Appendix F, less the special operator characters listed in Section 5.1.

many matching ‘>’ or ‘|’ characters as appropriate. Thus, for example, “<<//” matches “\>>”, and “|\\|” matches “///|”. (As in case 2 above, we may allow tokens with mixtures of ‘/’ and ‘\’ in the future.)

5. Finally, any number of ‘*’ (U+002A) or ‘.’ (U+002E) characters may be placed within any of the above multicharacter sequences, except those that contain ‘(’ or ‘)’, as long as no ‘*’ or ‘.’ is the first or last character in the sequence, and no ‘*’ or ‘.’ characters are adjacent. The rule for matching is as above, except that in addition, the positions of the ‘*’ and the ‘.’ characters must match from the outside in.

Note that some of the character sequences described above cannot occur in programs after ASCII conversion. For example, “| |” is converted to ‘||’ (U+2016). Note also that [\ \] (which are converted to [[]]) are not operators; they play a special role in the syntax of Fortress, and their behavior cannot be redefined by a library.

5.14.2 Special Operators

Note that in the preceding discussion, a single operator character can be an operator token only if it is an ordinary operator character. In some cases, some of the special operator characters (and even some special non-operator characters) form part of an operator token. However, most of the special operator characters cannot be determined to be operators before parsing the program because they are also used for various parts of Fortress syntax. The one exception is ‘^’: if an occurrence of this character is not part of a superscripted postfix operator, then it is an operator token by itself: the special *superscripting operator*. This operator is always an infix operator, and it is a static error if it appears in a context in which a prefix or postfix operator is expected.

Every other special operator character, when not part of an operator token, is a special token that may be used as an operator. There is also a special *juxtaposition operator* (described in Section 16.7), which is also always infix, but this operator is a reserved word rather than an operator token. Occurrences of this operator are determined by the Fortress grammar. The reserved words *in*, *cubed*, *cubic*, *in*, *inverse*, *per*, *square*, and *squared*, are used as operators on units.

5.15 Identifiers

A word is an identifier if it begins with a letter and is not a reserved word, an operator, or all or part of a numeral.

5.16 Special Tokens

Every special character (operator or non-operator) that is not part of a token (or within a comment) as described above is a *special token* by itself. The special operator characters may be operators in the appropriate context.

5.17 Rendering of Fortress Programs

In order to more closely approximate mathematical notation, Fortress mandates standard rendering for various input elements, particularly for numerals and identifiers, as specified in this section. In the remainder of this specification, programs are presented formatted unless stated otherwise.

5.17.1 Fonts

Throughout this section, we refer to different fonts or styles in which certain characters are rendered, with names suggestive of their appearance.

- roman
- italic
- math (often identical to italic)
- script
- fraktur
- sans-serif
- italic sans-serif
- monospace
- double-struck

Additionally, the following fonts may be specified to be bold: roman, italic, script, fraktur, sans-serif, italic sans-serif.

However, a particular environment may substitute different fonts either because of local practice or because the desired fonts are not available.

5.17.2 Numerals

A numeral is rendered in roman type, with the radix, if present, as a subscript.

27	<i>is rendered as</i>	27
7FFF ₁₆	<i>is rendered as</i>	7FFF ₁₆
10101101_TWO	<i>is rendered as</i>	10101101_TWO
37X8E2 ₁₂	<i>is rendered as</i>	37X8E2 ₁₂
deadbeef_SIXTEEN	<i>is rendered as</i>	deadbeef_SIXTEEN
dead.beef ₁₆	<i>is rendered as</i>	dead.beef ₁₆
3.143159265	<i>is rendered as</i>	3.143159265
3.11037552 ₈	<i>is rendered as</i>	3.1137552 ₈
3.243f6b ₁₆	<i>is rendered as</i>	3.243f6b ₁₆
11.001001000011111101101010 ₂	<i>is rendered as</i>	11.001001000011111101101010 ₂

Note: the elegant way to write Avogadro's number is `6.02 TIMES 10^23`, which is not a single token but is a constant expression; its rendered form is 6.02×10^{23} .

5.17.3 Identifiers

Fortress has rather complicated rules for rendering an identifier; as in other parts of Fortress, the rules are complicated so that the simple cases will be very simple, but also so that difficult cases of interest will be possible.

It is conventional in mathematical notation to make use of variables, particularly single-letter variables, in a number of different fonts or styles, with italic being the most common, then boldface, and roman: *a*, **b**, c. Frequently such variables are also decorated with accents and subscripts: \bar{p} , q' , \hat{r} , T_{\max} , \vec{u} , \mathbf{v}_x , w_{17} , \bar{z}'_{17} . Fortress provides conventions for typing such variables using plain ASCII characters: for example, the unformatted presentations of these same variables are a, _b, c_, p_bar, q' or q_prime, r_hat, T_max, _u_vec, _v_x, w17,

and `z17_bar'`. The rules are also intended to accommodate the typical use of multicharacter variable names for computer programming, such as *count*, *isUpperCase*, and `Boolean`.

The most important rules of thumb are that simple variables are usually italic *z* (*z*), a leading underscore usually means boldface font **z** (**_z**), a trailing underscore usually means roman font *z* (*z_*), and a doubled capital letter means double-struck (or “blackboard bold”) font \mathbb{Z} (\mathbb{ZZ}). However, mixed-case variable names that begin with a capital letter, which are usually used as names of types, are rendered in roman font even if there is no trailing underscore.

The detailed rules are described in Appendix D.

5.17.4 Other Formatting Rules

Reserved words are rendered in monospace, except that the reserved words that are used as operators on units, namely *cubed*, *cubic*, *in*, *inverse*, *per*, *square*, and *squared*, are rendered in roman type. Operator words are rendered in monospace. Comments are rendered in roman font. Any character within a character literal or string literal is rendered in monospace if possible. Delimiters of syntax expanders (described in Section 36.1) are rendered in monospace.

Chapter 6

Declarations

Declarations introduce *named entities*; we say that a declaration *declares* an entity and a name by which that entity can be referred to, and that the declared name *refers to* the declared entity. As discussed later, there is not a one-one correspondence between declarations and named entities: some declarations declare multiple named entities, some declare multiple names, and some named entities are declared by multiple declarations.

Some declarations contain other declarations. For example, a trait declaration may contain method declarations, and a function declaration may contain parameter declarations.

The positions in which a declaration may legally appear are determined by the nonterminal *Decl* in the simplified Fortress grammar in Appendix G.

6.1 Kinds of Declarations

Syntax:

```
Decl ::= TraitDecl
      | ObjectDecl
      | VarDecl
      | FnDecl
      | DimUnitDecl
      | TypeAlias
      | TestDecl
      | PropertyDecl
```

There are two kinds of declarations: *top-level declarations* and *local declarations*.

Top-level declarations occur at the top level of a component, not within any other declaration or expression. A top-level declaration is one of the following:¹

- trait declarations (see Chapter 9);
- object declarations (see Chapter 10), which may be singleton declarations or constructor declarations;
- top-level variable declarations (see Section 6.2);
- top-level function declarations (see Chapter 12), including top-level operator declarations (see Chapter 16);

¹The Fortress component system, defined in Chapter 22, includes declarations of *components* and *APIs*. Because component names are not used in a Fortress program and API names are used only in qualified names and `import` and `export` statements, we do not discuss them in this chapter.

- dimension declarations (see Chapter 18);
- unit declarations (see Chapter 18);
- top-level type aliases (see Section 8.9);
- test declarations (see Chapter 19);
- top-level property declarations; (see Chapter 19)

Local declarations occur in another declaration or in some expression (or both). These may be one of the following:

- field declarations (see Section 10.2), which occur in object declarations and object expressions, and include field declarations in the parameter list of a constructor declaration;
- method declarations (see Section 9.2), which occur in trait and object declarations and object expressions;
- coercion declarations (see Chapter 17), which occur in trait and object declarations;
- local variable declarations (see Section 6.3), which occur in block expressions;
- local function declarations (see Section 6.4), which occur in block expressions;
- local property declarations (see Chapter 19), which occur in trait and object declarations and object expressions;
- labeled blocks (see Section 13.12), which are expressions;
- static-parameter declarations, which may declare type parameters, `nat` parameters, `int` parameters, `bool` parameters, `dim` parameters, `unit` parameters, `opr` parameters, or `ident` parameters (see Chapter 11), and occur in static-parameter lists of trait and object declarations, top-level type aliases, top-level function declarations, and method declarations
- hidden-type-variable declarations, which occur in `where` clauses (see Section 11.6) of trait and object declarations, top-level function declarations, and method declarations;
- type aliases in `where` clauses, of trait and object declarations, top-level function declarations, and method declarations;
- (value) parameter declarations, which occur in parameter lists of constructor declarations, top-level function declarations, method declarations, local function declarations, and function expressions (but do not include non-transient parameter declarations in the parameter list of a constructor declaration, which are field declarations)

Some declarations are syntactic sugar for other declarations. Throughout this chapter, we consider declarations after they have been desugared. Thus, apparent field declarations in trait declarations are actually method declarations (as described in Section 9.2), and a dimension and unit declaration may desugar into several separate declarations (as described in Section 35.3). After desugaring, the kinds of declarations listed above are disjoint.

In addition to these explicit declarations, there are two cases in which names are declared implicitly:

- the special name `self` is implicitly declared as a parameter of dotted methods whose declarations do not provide an explicit declaration for the `self` parameter (see Section 9.2 for details); and
- the name `result` is implicitly declared as a variable for the `ensures` clause of a contract (see Section 12.4 for details).

Trait declarations, object declarations, top-level type aliases, type-parameter declarations, and hidden-type-variable declarations are collectively called *type declarations*; they declare names that refer to *types* (see Chapter 8). Dimension declarations and `dim`-parameter declarations are *dimension declarations*, and unit declarations and `unit`-parameter declarations are *unit declarations*. Constructor declarations, top-level function declarations, method declarations, and local function declarations are collectively called *functional declarations*. Singleton declarations, top-level variable declarations, field declarations, local variable declarations (including implicit declarations of `result`) and (value)

parameter declarations (including implicit declarations of `self`) are collectively called *variable declarations*. Static-parameter declarations and hidden-type-variable declarations are collectively called *static-variable declarations*. Note that static-variable declarations are disjoint from variable declarations.

The groups of declarations defined in the previous paragraph are neither disjoint nor exhaustive. For example, labeled blocks are not included in any of these groups, and an object declaration is both a type declaration and either a function or variable declaration, depending on whether it is singleton.

Most declarations declare a single name given explicitly in the declaration (though, as discussed in Section 7.1, they may declare this name in multiple namespaces). There is one exception: wrapped field declarations (described in Section 9.3) in object declarations and object expressions declare both the field name and names for methods provided by the declared type of the field.

Method declarations in a trait may be either *abstract* or *concrete*. Abstract declarations do not have bodies; concrete declarations, sometimes called *definitions*, do.

6.2 Top-Level Variable Declarations

Syntax:

```

VarDecl      ::=  VarWTypes InitVal
              |  VarWoTypes = Expr
              |  VarWoTypes : TypeRef ... InitVal
              |  VarWoTypes : SimpleTupleType InitVal
VarWTypes    ::=  VarWType
              |  ( VarWType( , VarWType)+ )
VarWType     ::=  VarMods? Id IsType
VarWoTypes   ::=  VarWoType
              |  ( VarWoType( , VarWoType)+ )
VarWoType    ::=  VarMods? Id
InitVal      ::=  (= | :=) Expr
SimpleTupleType ::=  ( TypeRef , TypeRefList )
TypeRefList  ::=  TypeRef( , TypeRef)*
VarMods      ::=  VarMod+
VarMod       ::=  var | UniversalMod
IsType       ::=  : TypeRef

```

A *variable*'s name can be any valid Fortress identifier. There are three forms of variable declarations. The first form:

```
id : Type = expr
```

declares *id* to be an immutable variable with static type *Type* whose value is computed to be the value of the *initializer* expression *expr*. The static type of *expr* must be a subtype of *Type*.

The second (and most convenient) form:

```
id = expr
```

declares *id* to be an immutable variable whose value is computed to be the value of the expression *expr*; the static type of the variable is the static type of *expr*.

The third form:

```
var id : Type = expr
```

declares id to be a mutable variable of type `Type` whose initial value is computed to be the value of the expression $expr$. As before, the static type of $expr$ must be a subtype of `Type`. The modifier `var` is optional when `:=` is used instead of `=` as follows:

```
var? id : Type := expr
```

In short, immutable variables are declared and initialized by `=` and mutable variables are declared and initialized by `:=` except when they are declared as the third form above with the modifier `var`.

All forms can be used with *tuple notation* to declare multiple variables together. Variables to declare are enclosed in parentheses and separated by commas, as are the types declared for them:

```
(id, id)+ : (Type, Type)+
```

Alternatively, the types can be included alongside the respective variables, optionally eliding types that can be inferred from context (see Chapter 20 for a discussion of type inference in Fortress):

```
(id(: Type)?, id(: Type)?)+
```

Alternatively, a single type followed by `...` can be declared for all of the variables:

```
(id, id)+ : Type ...
```

This notation is especially helpful when a function application returns a tuple of values.

The initializer expressions of top-level variable declarations can refer to variables declared later in textual order. Evaluation of top-level initializer expressions cannot call functionals with the modifier `io`.

Here are some simple examples of variable declarations:

```
π = 3.141592653589793238462643383279502884197169399375108209749445923078
```

declares the variable π to be an approximate representation of the mathematical object π . It is also legal to write:

```
π : ℝ64 = 3.141592653589793238462643383279502884197169399375108209749445923078
```

This definition enforces that π has static type `ℝ64`.

The following example declares multiple variables using tuple notation:

```
var (x, y) : ℤ64 ... = (5, 6)
```

The following three declarations are equivalent:

```
(x, y, z) : (ℤ64, ℤ64, ℤ64) = (0, 1, 2)
(x: ℤ64, y: ℤ64, z: ℤ64) = (0, 1, 2)
(x, y, z) : ℤ64 ... = (0, 1, 2)
```

6.3 Local Variable Declarations

Syntax:

```

LocalVarDecl ::= LocalVarWTypes InitVal
              | LocalVarWTypes
              | LocalVarWoTypes = Expr
              | LocalVarWoTypes : TypeRef ... InitVal?
              | LocalVarWoTypes : SimpleTupleType InitVal?
LocalVarWTypes ::= LocalVarWType
                | ( LocalVarWType( , LocalVarWType)+ )
LocalVarWType ::= var ? Id IsType
LocalVarWoTypes ::= LocalVarWoType
                 | ( LocalVarWoType( , LocalVarWoType)+ )
LocalVarWoType ::= var ? Id
                 | Unpasting

```

Variables can be declared within block expressions (described in Section 13.11) via the same syntax as is used for top-level variable declarations (described in Section 6.2) except that local variables must not include modifiers and additional syntax is allowed as follows:

- The form:

```
var? id : Type
```

declares a variable without giving it an initial value (where mutability is determined by the presence of the `var` modifier). It is a static error if a variable is referred to before it has been given a value; an immutable variable is initialized by another variable declaration and a mutable variable is initialized by assignment. It is also a static error if an immutable variable is initialized more than once. Whenever a variable bound in this manner is assigned a value, the type of that value must be a subtype of its declared type. This form allows declaration of the types of variables to be separated from definitions, and it allows programmers to delay assigning to a variable before a sensible value is known. In the following example, the declaration of the type of a variable and its definition are separated:

```

π : Float
π = 3.141592653589793238462643383279502884197169399375108209749445923078

```

- Special syntax for declaring local variables as parts of a matrix is provided as described in Section 6.5.

6.4 Local Function Declarations

Syntax:

```

LocalFnDecl ::= LocalFnMods? FnHeaderFront FnHeaderClause = Expr
LocalFnMod  ::= atomic | io
LocalFnMods ::= LocalFnMod+

```

Functions can be declared within block expressions (described in Section 13.11) via the same syntax as is used for top-level function declarations (described in Chapter 12) except that locally declared functions must not include the modifiers `private` and `test`. As with top-level function declarations, locally declared functions in a single scope are allowed to be overloaded and mutually recursive.

6.5 Matrix Unpasting

Syntax:

```

Unpasting ::= [ UnpastingElems ]
UnpastingElems ::= UnpastingElem RectSeparator UnpastingElems
| UnpastingElem
UnpastingElem ::= Id ( [ UnpastingDim ] )?
| Unpasting
UnpastingDim ::= ExtentRange ( × ExtentRange )+
ExtentRange ::= StaticArg? # StaticArg?
| StaticArg? : StaticArg?
| StaticArg
RectSeparator ::= ; +
| Whitespace

```

Matrix unpasting is an extension of local variable declaration syntax as a shorthand for breaking a matrix into parts. On the left-hand side of a declaration, what looks like a matrix pasting of unbound variables is actually a declaration of several new variables. This syntax serves to break the right-hand side into pieces and bind the pieces to the variables. Matrix unpastings are concise, eliminate several opportunities for fencepost errors, guarantee unaliased parts, and avoid overspecification of how the matrix should be taken apart.

The motivating example for matrix unpasting is cache-oblivious matrix multiplication. The general plan in a cache-oblivious algorithm is to break the input apart on its largest dimension, and recursively attack the resulting smaller and more compact problems.

```

mm [ nat m, nat n, nat p ] ( left :  $\mathbb{R}^{m \times n}$ , right :  $\mathbb{R}^{n \times p}$ , result :  $\mathbb{R}^{m \times p}$  ) : () =
  case largest of
    1 ⇒ result0,0 += ( left0,0 right0,0 )
    m ⇒ [ lefttop
           leftbottom ] = left
           [ resulttop
             resultbottom ] = result
           t1 = spawn mm ( lefttop, right, resulttop )
                mm ( leftbottom, right, resultbottom )
           t1.wait()
    p ⇒ [ rightleft rightright ] = right
           [ resultleft resultright ] = result
           t1 = spawn mm ( left, rightleft, resultleft )
                mm ( left, rightright, resultright )
           t1.wait()
    n ⇒ [ leftleft leftright ] = left
           [ righttop
             rightbottom ] = right
           mm ( leftleft, righttop, result )
           mm ( leftright, rightbottom, result )
  end

```

In unpasting, the element syntax is slightly enhanced both to permit some specification of the split location and to receive information about the split that was performed. For example, perhaps only the upper left square of a matrix is interesting. The programmer can annotate bounds to the square unpasted element:

```

foo [ nat m, nat n ] ( A :  $\mathbb{R}^{m \times n}$  ) : () =
  if m < n then
    [ squareShapem × m rest ] = A
    ...
  elif m > n then

```

```

    [ squareShapen×n
      rest ] = A
    ...
else (* A already square *)
    squareShape = A
    ...
end

```

The types of the elements of the newly declared matrix variables on the left-hand side of an unpasting are inferred (trivially) to be the type of the elements on the right-hand side.

If an unpasting into explicitly sized pieces does not exactly cover the right-hand-side matrix, an `UnpastingException` is thrown.

Each element of the left-hand-side of unpasting includes an optional *extent specification*. An extent specification `low # num` describes the indexing and the size of the given part of the matrix. The lower extent must be bound, either before the unpasting, or earlier (left-or-above) in the unpasting. For example, suppose that an algorithm chooses to break a matrix into 4 pieces, but retain the original indices for each piece:

```

bar[[nat p, nat q]](X : ℝr0 # p × c0 # q) : () = do
  [ Ar0 # m      × c0 # n  Br0 # m      × c0 + n # q - n
    Cr0 + m # p - m × c0 # n  Dr0 + m # p - m × c0 + n # q - n ] = X
  ...
end

```

Unpasting does not directly support non-uniform decomposition, and does not provide any sort of constraint satisfaction between the extents of the parts. For example, the following decomposition is not legal because it constrains the split sizes to be equal with respect to unbound `nat` parameters:

```

(* Not allowed! *)
fubar[[nat m, nat n]](X : ℝm × n) : () = do
  (* p and q unbound *)
  [ Ap × q  Bp × q
    Cp × q  Dp × q ] = X
  ...
end

```

To get this effect, the programmer should compute the constrained values:

```

fubar[[nat m, nat n]](X : ℝ2m × 2n) : () = do
  [ Am × n  Bm × n
    Cm × n  Dm × n ] = X
  ...
end

```

Some non-uniform unpastings can be obtained with composition, which can be expressed either by repeated unpasting:

```

unequalRows[[nat m, nat n]](X : ℝ4m × 2n) = do
  [ c14m × n  c24m × n ] = X
  [ Am × n
    C3m × n ] = c1
  [ B3m × n
    Dm × n ] = c2
  ...
end

```


or simply by nesting matrices in the unpasting:

```
unequalRows[[nat m, nat n]](X :  $R^{2m \times 4n}$ ) = do  
  [[Am × n Bm × 3n]  
   [Cm × 3n Dm × n]] = X  
  ...  
end
```

Chapter 7

Names

Names¹ are used to refer to certain kinds of entities in a Fortress program. Names may be simple or qualified. A simple name is either an identifier or an operator. An operator may be an operator token, a special token corresponding to a special operator character, or a matching pair of enclosing operator tokens. A qualified name consists of an API name followed by “.”, followed by an identifier, where an API name consists of a sequence of identifiers separated by “.” tokens. Note that operators cannot be qualified. Except in Section 7.3, we consider only simple names in this chapter.

Simple names are typically introduced by declarations, which bind the name to an entity. In some cases, the declaration is implicit. Every declaration has a scope, in which the declared name can be used to refer to the declared entity.

7.1 Namespaces

Fortress supports three namespaces, one for types, one for values, and one for labels. (If we consider the Fortress component system, there is another namespace for APIs.) These namespaces are logically disjoint: names in one namespace do not conflict with names in another.

Type declarations, of course, declare names in the type namespace. Function and variable declarations declare names in the value namespace. (This implies that object names are declared in both the type and value namespaces.) Labeled blocks declare names in the label namespace. Although they are not all type declarations, all the static-variable declarations declare names in the type namespace, as do dimension declarations. In addition, `nat` parameters, `int` parameters, `bool` parameters, `unit` parameters, `opr` parameters, and `ident` parameters are also declared in the value namespace. Section 11.5 describes `opr` parameters and `ident` parameters in more detail.

A reference to a name is resolved to the entity that the name refers to the namespace appropriate to the context in which the reference occurs. For example, a name refers to a label if and only if it occurs immediately following the reserved word `exit`. It refers to a type if and only if it appears in a type context (described in Chapter 8). Otherwise, it refers to a value.

7.2 Reach and Scope of a Declaration

In this section, we define the *reach* and *scope* of a declaration, which determine where a declared name may be used to refer to the entity declared by the declaration. It is a static error for a reference to a name to occur at any point in a program at which the name is not in scope in the appropriate namespace (as defined below), except immediately after

¹A name described in this chapter does not syntactically correspond to the nonterminal *Name* in the Fortress grammar, defined in Appendix G.

the ‘.’ of a dotted field access or dotted method invocation when the name is the name of the field or dotted method of the static type of the receiver expression, whose name must be in scope in the type namespace (see Sections 13.3 and 13.4).

We first define a declaration’s *reach*. The reach of a labeled block is the block itself. The reach of a functional method declaration is the component containing that declaration. A dotted method declaration not in an object expression or declaration must be in the declaration of some trait T , and its reach is the declaration of T and any trait or object declarations or object expressions that extend T ; that is, if the declaration of trait T contains a method declaration, and trait S extends trait T , then the reach of that method declaration includes the declaration of trait S . The reach of any other declaration is the smallest block strictly containing that declaration (i.e., not just the declaration itself). For example, the reach of any top-level declaration (including any imported declaration) is the component containing (or importing) that declaration; the reach of a field declaration is the enclosing object declaration or expression; the reach of a parameter declaration is the constructor declaration, functional declaration, or function expression in whose parameter list it occurs; and the reach of a local variable declaration is the smallest block in which that declaration occurs (recall from Chapter 3 that a local variable declaration always starts a new block). The reach of an implicit declaration of `self` for a dotted method declaration is the method declaration, and the reach of an implicit declaration of *result* for an `ensures` clause is the `ensures` clause. We say that a declaration *reaches* any point within its reach.

It is a static error for two (explicit) declarations with overlapping reaches to declare the same name, *even if the name is declared in different namespaces*, unless one of the following (disjoint) conditions holds:

- both declarations are functional declarations with the same reach;
- one is an abstract function declaration in the form of a top-level variable declaration and the other a top-level function declaration or a functional method declaration;
- both declarations are dotted method declarations that occur in different trait declarations;
- one declaration is a field declaration in an object declaration or expression and the other is a getter or setter declaration that is provided by (i.e., occurs in or is inherited by) that object declaration or expression;
- one declaration is a keyword-parameter declaration whose reach is strictly contained in the reach of the other declaration;
- one declaration is a field or dotted method declaration that is provided by (i.e., occurs in or is inherited by) some trait or object declaration and the other is a top-level declaration or a functional method declaration; or
- one declaration is a field or dotted method declaration that is provided by (i.e., occurs in or is inherited by) some object expression that is strictly contained in the reach of the other declaration, and the other declaration is not a dotted method declaration provided by a trait that is extended by the object expression.

If any of the first four conditions holds, or if one declaration is a field or method declaration that occurs in an object declaration or expression that inherits the other declaration (which therefore must be a method declaration), then the two declarations are *overloaded*, and subject to the overloading rules (see Chapter 33).

If two declarations with overlapping reaches declare the same name in the same namespace, and the declarations are not overloaded, then at any point that their reaches overlap, one declaration *shadows* the other for that name in that namespace; references at such points resolve to the entity declared by the shadowing declaration. We often elide mentioning the name and namespace when they are clear from context. Shadowing is permitted only in the following cases:

- In a trait or object declaration, any top-level declaration or functional method declaration is shadowed if it declares a name of a field or dotted method provided (declared or inherited) by the trait or object.
- In an object expression, any declaration in a block enclosing the expression is shadowed if it declares a name of a field or dotted method provided (declared or inherited) by the object expression unless the declaration is a method declaration of a trait extended by the object expression.

- In a method declaration that does not give an explicit name other than `self` for the self parameter, any declaration of `self` (including implicit declarations) in a block enclosing the field or method declaration is shadowed.
- In the `ensures` clause of a contract, any declaration of `result` in a block enclosing the `ensures` clause is shadowed.
- In a function or method declaration with keyword parameters, any declaration in a block enclosing the declaration is shadowed if it declares the name of any of the keyword parameters.
- Any immutable local variable declaration without an initial-value expression is shadowed if the variable is initialized later in the enclosing block.

We say that a name is *in scope* in a namespace at any point in the program within the reach of any declaration that declares that name in that namespace unless one of the following conditions holds:

- the declaration is shadowed at the program point for the name in that namespace;
- the declaration is a type alias (top-level or not), a dimension declaration or a unit declaration, and the program point is in the declaration;
- the declaration is a field, local variable or parameter declaration, and the program point is in the declaration or lexically precedes the declaration;
- the declaration is a (transient) parameter declaration of an object (constructor) declaration and the program point is in the body of a method declaration of that object declaration; or
- the declaration is a labeled block, and the program point is in a `spawn` expression in the labeled block.

Note that the last condition applies to the method names declared by a wrapped field declaration.

We say that the *scope* of a declaration for a name in a namespace consists of those points at which the name is in scope for the namespace and the declaration is not shadowed for that name and that namespace. Again, when it is clear from context, we may omit the name and namespace.

7.3 Qualified Names

Syntax:

$$\textit{DottedId} ::= \textit{Id}(. \textit{Id})^*$$

Fortress provides a component system in which the entities declared in a component are described by an API. A component may *import* APIs, allowing it to refer to these entities declared by the imported APIs. In some cases, references to these entities must be *qualified* by the API name. These qualified names can be used in any place that a simple name would be used had the entity been declared directly in the component rather than being imported. Note that qualified names are distinguished from simple names by the inclusion of a “.” token, so they never shadow, nor are they shadowed by, simple names. For further discussion on APIs and the component system, see Chapter 22.

Chapter 8

Types

Fortress provides several kinds of types: trait types, tuple types, arrow types, `BottomType`, and other types provided in the Fortress standard libraries. Some types have names. Some types may be parameterized by types and values; we call these types *generic types*. Two types are identical if and only if they are the same kind and their names and arguments (if any) are identical. Types are related by several relationships as described in Section 8.1.

Syntactically, the positions in which a type may legally appear (*type context*) is determined by the nonterminal *TypeRef* in the Fortress grammar, defined in Appendix G.

8.1 Relationships between Types

Types in Fortress may be related by a subtyping relation, an exclusion relation, or a coercion.

A *subtyping* relation is reflexive, transitive, and antisymmetric, and is defined by the `extends` clause of trait and object declarations and object expressions. Every expression has a *static type*. Every value has a *runtime type* (dynamic type). Fortress programs are checked before they are executed to ensure that if an expression e evaluates to a value v , the runtime type of v is a subtype of the static type of e . Sometimes we abuse terminology by saying that an expression has the runtime type of the value it evaluates to (see Section 4.2 for a discussion about evaluation of expressions). Thus, in the execution of a valid Fortress program, an expression's runtime type is always a subtype of its static type. We say that a value is *an instance of* its runtime type and of every supertype of its runtime type. Every type is a subtype of `Any`; immediate subtypes of `Any` comprises of `Tuple`, `()`, and `Object`. For types T and U , we write $T \preceq U$ when T is a subtype of U , and $T \prec U$ when $T \preceq U$ and $T \neq U$.

Fortress defines an *exclusion* relation between types, which relates two disjoint types: no value can have a type that is a subtype of two types that exclude each other. The exclusion relation is irreflexive and symmetric, and is defined by the `excludes` and `comprises` clauses of trait declarations, and what is implied from these by the subtyping relation (including the fact that object trait types have no strict subtypes, and so exclude all types other than their supertypes). For example, suppose the following:

```
trait S comprises {U, V} end
trait T comprises {V, W} end
object U extends S end
object V extends {S, T} end
object W extends T end
```

Because of the `comprises` clauses of S and T and the fact that U , V , and W are objects, S and T exclude each other. We write $T \diamond U$ if T excludes U . If a type excludes another type, it excludes all its subtypes as well:

$$T \diamond U \implies \forall T' \preceq T : T' \diamond U.$$

Fortress also allows *coercion* between types (see Chapter 17). A coercion from T to U is defined in the declaration of U . We write $T \rightarrow U$ if U defines a coercion from T . We say that T can be coerced to U , and write $T \rightsquigarrow U$, if U defines a coercion from T or any supertype of T : $T \rightsquigarrow U \iff \exists T' : T \preceq T' \wedge T' \rightarrow U$.

The Fortress type hierarchy is acyclic with respect to both subtyping and coercion relations except for the following:

- There exists a bidirectional coercion between two tuple types if and only if they have the same sorted form.

8.2 Trait Types

Syntax:

$$\textit{TypeRef} ::= \textit{TraitType}$$

Traits are declared by trait declarations (described in Chapter 9). A trait has a *trait type* of the same name. A significant portion of Fortress types are trait types.

8.3 Object Trait Types

Named objects are declared by object declarations (described in Chapter 10) and anonymous objects are described by object expressions (described in Section 13.9). A named object has an *object trait type* of the same name and an anonymous object has an anonymous object trait type. An object trait type is a special kind of trait type. An object trait type extends all of the declared supertraits of the object. No other objects can have the object trait type and no trait type can extend an object trait type (i.e., an object trait type implicitly has the `comprises` clause, “`comprises {...}`”).

8.4 Tuple Types

Syntax:

$$\begin{aligned} \textit{TypeRef} & ::= \textit{TupleType} \\ \textit{TupleType} & ::= ((\textit{TypeRef} \textit{,})^* (\textit{TypeRef} \dots \textit{,})? \textit{KeywordType}(\textit{,} \textit{KeywordType})^*) \\ & \quad | ((\textit{TypeRef} \textit{,})^* \textit{TypeRef} \dots) \\ & \quad | \textit{SimpleTupleType} \\ \textit{KeywordType} & ::= \textit{Id} = \textit{TypeRef} \\ \textit{SimpleTupleType} & ::= (\textit{TypeRef} \textit{,} \textit{TypeRefList}) \end{aligned}$$

A tuple is an ordered sequence of keyword-value pairs. See Section 13.27 for a discussion of tuple expressions. A tuple type consists of a parenthesized, comma-separated list of element types where each element type is one of the following kinds:

- A plain type “ T ”
- A varargs type “ $T \dots$ ”
- A keyword-type pair “ $\textit{identifier} = T$ ”

The following restrictions apply: No two keyword-type pairs may have the same keyword. No keyword-type pair may precede a plain type. No varargs type may follow a keyword-type pair or precede a plain type. There must be at least one element type. If there is exactly one element type, it must be a varargs type or a keyword-type pair (because “ (T) ” is simply a type in parentheses, not a tuple type). Also, there can be at most one element type with varargs type.

An element type in tuple type X corresponds to one in tuple type Y if and only if:

- both are plain types in the same position,
- both are varargs types, or
- both are keyword-type pairs with the same keyword.

Every tuple type is a subtype of `Tuple` which is a subtype of `Any`. Tuple types are not subtypes of `Object`. They cannot be extended by other trait types. Tuple types are covariant; a tuple type X is a subtype of tuple type Y if and only if:

- the correspondence between their element types is bijective;
- for each element type in X , the type in the element type is a subtype of the type in the corresponding element type in Y ; and
- the keyword-type pairs in X and Y appear in the same order.

Note that, unlike record types in some other programming languages, the tuple type $(foo = P, bar = Q, baz = R)$ is not a subtype of $(foo = P, bar = Q)$, nor is (P, Q, R) a subtype of (P, Q) . While (\mathbb{Z}, \mathbb{Z}) is not a subtype of $(\mathbb{Z} \dots)$, there exists a coercion from the former to the latter (as described in Section 17.7).

For every tuple type X there is a tuple type X' that is the “sorted form” of the type, created by simply reordering the keyword-type pairs so that their keywords are in lexicographically ascending order. X' may be the same as X (as, for example, if X contains fewer than two keyword-type pairs). There is a coercion from tuple type X to tuple type Y if and only if X and Y have the same sorted form.

A tuple type excludes any nontuple type other than `Any`. Two tuple types exclude each other unless the correspondence between their element types is bijective. Two tuple types with a bijective correspondence between their element types exclude each other if either any type in an element type in one excludes the type in the corresponding element type in the other, or their keyword-type pairs do not appear in the same order.

Intersection of nonexclusive tuple types are defined elementwise; the intersection of nonexclusive tuple type X and Y is a tuple type with exactly corresponding elements, where the type in each element type is the intersection of the types in the corresponding element types of X and Y . Note that intersection of any exclusive types is `BottomType` as described in Section 8.6.

8.5 Arrow Types

Syntax:

```
TypeRef ::= ArrowType
ArrowType ::= TypeRef → TypeRef Throws?
```

Functions can be passed as arguments and returned as values. See Chapter 12 for a discussion of functions. The types of function values are called *arrow types*. Every arrow type is a subtype of `Object`. Arrow types are not trait types. They cannot be extended by other trait types. Syntactically, an arrow type consists of the type of a parameter to the function followed by the token `→`, followed by the type of a return value, and optionally a `throws` clause which specifies thrown checked exceptions. Here are some examples:

```
(ℝ64, ℝ64) → ℝ64
ℕ → (ℕ, ℕ) throws IOException
(String, ℕ ..., p = Printer) → ℕ
```

Parameter types are contravariant but return types are covariant; arrow type “ $A \rightarrow B$ throws C ” is a subtype of arrow type “ $D \rightarrow E$ throws F ” if and only if:

- D is a subtype of A and

- B is a subtype of E and
- for all X in C , there exists Y in F such that X is a subtype of Y .

Coercion between arrow types are described in Section 17.7.

An arrow type excludes any nonarrow type other than `Any`. However, arrow types do not exclude other arrow types because of overloading as described in Chapter 33.

8.6 Bottom Type

Fortress provides a special *bottom type*, `BottomType`, which is an uninhabited type. No value in Fortress has the bottom type; `throw` and `exit` expressions have the bottom type. The bottom type is a subtype of every type. Intersection of any exclusive types is the bottom type.

8.7 Types in the Fortress Standard Libraries

The Fortress standard libraries define simple standard types for literals such as `BooleanLiteral[b]`, `()` (pronounced “void”), `Character`, `String`, and `Numeral[n, m, r, v]` for appropriate values of b , n , m , r , and v (See Section 13.1 for a discussion of Fortress literals). Moreover, there are several simple standard numeric types. These types are mutually exclusive; no value has more than one of them. Values of these types are immutable.

The numeric types share the common supertype `Number`. Fortress includes types for arbitrary-precision integers (of type `Z`), their unsigned equivalents (of type `N`), rational numbers (of type `Q`), fixed-size representations for integers including the types `Z8`, `Z16`, `Z32`, `Z64`, `Z128`, their unsigned equivalents `N8`, `N16`, `N32`, `N64`, `N128`, floating-point numbers (described below), intervals (of type `Interval[X]`, abbreviated as `<X>`, where X can be instantiated with any number type), and imaginary and complex numbers of fixed size (in rectangular form with types `Cn` for $n = 16, 32, 64, 128, 256$ and polar form with type `Polar[X]` where X can be instantiated with any real number type).

For floating-point numbers, Fortress supports types `R32` and `R64` to be 32 and 64-bit IEEE 754 floating-point numbers respectively, and defines two functions on types: `Double[F]` is a floating-point type twice the size of the floating-point type F , and `Extended[F]` is a floating-point type sufficiently larger than the floating-point type F to perform summations of “reasonable” size.¹

The Fortress standard libraries also define other simple standard types such as `Any`, `Object`, `Exception`, `Boolean`, and `BooleanInterval` as well as low-level binary data types such as `LinearSequence`, `HeapSequence`, and `BinaryWord`. See Parts III and V for discussions of the Fortress standard libraries.

8.8 Intersection and Union Types

For every finite set of types, there is a type denoting a unique *intersection* of those types. The intersection of a set of types S is a subtype of every type $T \in S$ and of the intersection of every subset of S . There is also a type denoting a unique *union* of those types. The union of a set of types S is a supertype of every type $T \in S$ and of the union of every subset of S . Neither intersection types nor union types are first-class types; they are used solely for type inference (as described in Chapter 20) and they cannot be expressed directly in programs.

The intersection of a set of types S is equal to a named type U when any subtype of every type $T \in S$ and of the intersection of every subset of S is a subtype of U . Similarly, the union of a set of types S is equal to a named type U when any supertype of every type $T \in S$ and of the union of every subset of S is a supertype of U . For example:

¹ This formulation of floating-point types follows a proposal under consideration by the IEEE 754 committee.


```

trait S comprises {U, V} end
trait T comprises {V, W} end
trait U extends S excludes W end
trait V extends {S, T} end
trait W extends T end

```

because of the `comprises` clauses of S and T and the `excludes` clause of U , any subtype of both S and T must be a subtype of V . Thus, $V = S \cap T$.

Intersection types (denoted by \cap) possess the following properties:

- Commutativity: $T \cap U = U \cap T$.
- Associativity: $S \cap (T \cap U) = (S \cap T) \cap U$.
- Subsumption: If $S \preceq T$ then $S \cap T = S$.
- Preservation of shared subtypes: If $T \preceq S$ and $T \preceq U$ then $T \preceq S \cap U$.
- Preservation of supertype: If $S \preceq T$ then $\forall U. S \cap U \preceq T$.
- Distribution over union types: $S \cap (T \cup U) = (S \cap T) \cup (S \cap U)$.

Union types (denoted by \cup) possess the following properties:

- Commutativity: $T \cup U = U \cup T$.
- Associativity: $S \cup (T \cup U) = (S \cup T) \cup U$.
- Subsumption: If $S \preceq T$ then $S \cup T = T$.
- Preservation of shared supertypes: If $S \preceq T$ and $U \preceq T$ then $S \cup U \preceq T$.
- Preservation of subtype: If $T \preceq S$ then $\forall U. T \preceq S \cup U$.
- Distribution over intersection types: $S \cup (T \cap U) = (S \cup T) \cap (S \cup U)$.

8.9 Type Aliases

Syntax:

```

TypeAlias ::= type Id StaticParams? = TypeRef

```

Fortress allows names to serve as aliases for more complex type instantiations. A *type alias* begins with `type` followed by the name of the alias type, followed by optional static parameters, followed by `=`, followed by the type it stands for. Parameterized type aliases are allowed but recursively defined type aliases are not. Here are some examples:

```

type IntList = List[[Z64]]
type BinOp = Float × Float → Float
type SimpleFloat[[nat e, nat s]] = DetailedFloat[[Unity, e, s, false, false, false, false, true]]

```

All uses of type aliases are expanded before type checking. Type aliases do not define new types nor nominal equivalence relations among types.

Chapter 9

Traits

Traits are declared by trait declarations. Traits define new named types. A trait specifies a collection of *methods* (described in Section 9.2). One trait can extend others, which means that it inherits the methods from those traits, and that the type defined by that trait is a subtype of the types of traits it extends.

9.1 Trait Declarations

Syntax:

```
TraitDecl ::= TraitHeader GoInATrait? end
TraitHeader ::= TraitMods? trait Id StaticParams? Extends? TraitClauses?
TraitClauses ::= TraitClause+
TraitClause ::= Excludes
                | Comprises
                | Where
GoInATrait ::= GoFrontInATrait GoBackInATrait?
                | GoBackInATrait
GoFrontInATrait ::= GoesFrontInATrait+
GoesFrontInATrait ::= AbsFldDecl
                | GetterSetterDecl
                | PropertyDecl
GoBackInATrait ::= GoesBackInATrait+
GoesBackInATrait ::= MdDecl
                | PropertyDecl
TraitMods ::= TraitMod+
TraitMod ::= value | UniversalMod
UniversalMod ::= private | test
Extends ::= extends TraitTypes
Excludes ::= excludes TraitTypes
Comprises ::= comprises ComprisingTypes
TraitTypes ::= TraitType
                | { TraitTypeList }
TraitTypeList ::= TraitType( , TraitType)*
```

```

ComprisingTypes ::= TraitType
                  | { ComprisingTypeList }
ComprisingTypeList ::= ...
                  | TraitType(, TraitType)* (, ...)?
TraitType ::= DottedId ([[StaticArgList]])?
              | { TypeRef  $\mapsto$  TypeRef }
              | < TypeRef >
              | TypeRef [ ArraySize? ]
              | TypeRef ^ StaticArg
              | TypeRef ^ ( ExtentRange (  $\times$  ExtentRange)* )
StaticArgList ::= StaticArg(, StaticArg)*
StaticArg ::= Number
              | Op
              | true
              | false
              | Unity
              | dimensionless
              | StaticArg + StaticArg
              | StaticArg  $\cdot$  StaticArg
              | StaticArg StaticArg
              | StaticArg / StaticArg
              | 1 / StaticArg
              | StaticArg ^ StaticArg
              | StaticArg per StaticArg
              | DUPreOp StaticArg
              | StaticArg DUPostOp
              | NOT StaticArg
              | StaticArg OR StaticArg
              | StaticArg AND StaticArg
              | StaticArg IMPLIES StaticArg
              | TypeRef
              | (StaticArg)
Number ::= IntLiteral
ArraySize ::= ExtentRange(, ExtentRange)*

```

Syntactically, a trait declaration starts with an optional sequence of modifiers followed by `trait`, followed by the name of the trait, an optional sequence of static parameters (described in Chapter 11), an optional set of *extended* traits, an optional set of *excluded* traits, an optional set of *comprises* on the trait, an optional `where` clause (described in Section 11.6), zero or more declarations of abstract fields, getter methods, and setter methods, and zero or more declarations of methods that are not getter or setter, and finally `end`. Property declarations (described in Section 19.6) can be freely commingled with the abstract field and method declarations. An `extends` clause comes first, if any, and `excludes`, `comprises`, and `where` clauses come in any order.

Each of `extends`, `excludes`, and `comprises` clauses consists of `extends`, `excludes`, and `comprises` respectively followed by a set of trait references separated by commas and enclosed in braces ‘{’ and ‘}’. If such a clause contains only one trait, the enclosing braces may be elided. A `comprises` clause may include “...”. A trait reference listed in the `comprises` clause is either a declared trait identifier (within the same component or an API imported by the component) or an abbreviated type for aggregate expressions (discussed in Section 13.27).

Every trait extends the trait `Object`. A trait with an `extends` clause extends every trait listed in its `extends` clause. If a trait T extends trait U , we call T a subtrait of U and U a supertrait of T . Extension is transitive; if T extends U it also extends all supertraits of U . The extension relation induced by a program is the smallest relation satisfying these conditions. This relation must form an acyclic hierarchy rooted at trait `Object`.

We say that trait T *strictly extends* trait U if and only if (i) T extends U and (ii) T is not U . We say that trait T *immediately extends* trait U if and only if (i) T strictly extends U and (ii) there is no trait V such that T strictly extends V and V strictly extends U . We call U an *immediate supertrait* of T and T an *immediate subtrait* of U .

A trait with an `excludes` clause excludes every trait listed in its `excludes` clause. If a trait T excludes a trait U , the two traits are mutually exclusive. No third trait can extend them both and neither can extend the other. A trait U can optionally have an `excludes` clause.

If a trait declaration of T includes a `comprises` clause and the `comprises` clause does not include "...", the trait must not be extended with immediate subtraits other than those that listed in its `comprises` clause. If the `comprises` clause of a trait T includes "...", any subtrait of T is not exposed by an API (as described in Section 22.3).

For example, the following trait declaration:

```
trait Catalyst extends Object
  self.catalyze(reaction: Reaction): ()
end
```

declares a trait `Catalyst` with no modifiers, no static parameters, no `excludes` clauses, no `comprises` clauses, and no `where` clauses. Trait `Catalyst` extends a trait named `Object`. A single method (named `catalyze`) is declared, which has a parameter of type `Reaction` and the return type `()`. The special name `self` is explicitly declared as a parameter. See Section 9.2 for details about when `self` is implicitly declared, and to which entity it refers.

The following example trait:

```
trait Molecule comprises { OrganicMolecule, InorganicMolecule }
  mass(): Mass
end
```

comprises of two traits: `OrganicMolecule` and `InorganicMolecule`. Therefore, the following trait declaration is not allowed:

```
(* Not allowed! *)
trait ExclusiveMolecule extends Molecule end
```

Traits `OrganicMolecule` and `InorganicMolecule` may be exclusive:

```
trait OrganicMolecule extends Molecule excludes InorganicMolecule end
trait InorganicMolecule extends Molecule end
```

`OrganicMolecule` and `InorganicMolecule` exclude each other, even though only `OrganicMolecule` has an `excludes` clause. For example, the following trait declaration is not allowed:

```
(* Not allowed! *)
trait InclusiveMolecule extends { InorganicMolecule, OrganicMolecule } end
```

A trait is allowed to have multiple immediate supertraits. The following trait has two immediate supertraits:

```
trait Enzyme extends { OrganicMolecule, Catalyst } end
```

9.2 Method Declarations

Syntax:

<i>MdDecl</i>	::=	<i>MdDef</i> <i>AbsMdDecl</i>
<i>MdDef</i>	::=	<i>FnMods?</i> <i>MdHeaderFront</i> <i>FnHeaderClause</i> = <i>Expr</i> <i>Coercion</i>
<i>AbsMdDecl</i>	::=	abstract ? <i>FnMods?</i> <i>MdHeaderFront</i> <i>FnHeaderClause</i>
<i>FnMods</i>	::=	<i>FnMod</i> ⁺
<i>FnMod</i>	::=	<i>LocalFnMod</i> <i>UniversalMod</i>
<i>LocalFnMods</i>	::=	<i>LocalFnMod</i> ⁺
<i>LocalFnMod</i>	::=	atomic io
<i>GetterSetterDecl</i>	::=	<i>GetterSetterDef</i> <i>AbsGetterSetterDecl</i>
<i>GetterSetterDef</i>	::=	<i>FnMods?</i> <i>GetterSetterMod</i> <i>MdHeaderFront</i> <i>FnHeaderClause</i> = <i>Expr</i>
<i>AbsGetterSetterDecl</i>	::=	abstract ? <i>FnMods?</i> <i>GetterSetterMod</i> <i>MdHeaderFront</i> <i>FnHeaderClause</i>
<i>GetterSetterMod</i>	::=	getter setter
<i>MdHeaderFront</i>	::=	(<i>Receiver</i> .)? <i>Id</i> <i>StaticParams?</i> <i>MdValParam</i> <i>OpHeaderFront</i>
<i>Receiver</i>	::=	<i>Id</i> self
<i>MdValParam</i>	::=	(<i>MdParams?</i>)
<i>MdParams</i>	::=	(<i>MdParam</i> ,)* (<i>Varargs</i> ,)? <i>MdKeyword</i> (, <i>MdKeyword</i>)* (<i>MdParam</i> ,)* <i>Varargs</i> <i>MdParam</i> (, <i>MdParam</i>)*
<i>MdKeyword</i>	::=	<i>MdParam</i> = <i>Expr</i>
<i>MdParam</i>	::=	<i>Param</i> self
<i>Param</i>	::=	<i>PlainParam</i>
<i>PlainParam</i>	::=	<i>BindId</i> <i>IsType?</i> <i>TypeRef</i>
<i>BindId</i>	::=	<i>Id</i> -
<i>FnHeaderClause</i>	::=	<i>IsType?</i> <i>FnClauses</i>
<i>IsType</i>	::=	: <i>TypeRef</i>
<i>FnClauses</i>	::=	<i>Throws?</i> <i>Where?</i> <i>Contract</i>
<i>Throws</i>	::=	throws <i>MayTraitTypes</i>
<i>MayTraitTypes</i>	::=	{ } <i>TraitTypes</i>

A trait declaration contains a set of method declarations. Syntactically, a method declaration begins with an optional sequence of modifiers followed by the method's name optionally prefixed by a *self* parameter, optional static parameters (described in Chapter 11), the value parameter with its (optionally) declared type, an optional type of a return value, an optional declaration of thrown checked exceptions (discussed in Chapter 14), an optional *where* clause (discussed in Section 11.6), a contract for the method (discussed in Section 9.4), and finally an optional body expression preceded by the token =. A *throws* clause does not include naked type variables. Every element in a *throws* clause is a subtype of *CheckedException*. A trait declaration may contain *coercions* discussed in Chapter 17.

Method declarations can include the following special modifiers:

getter: A method declaration with the modifier *getter* explicitly declares a getter method for a field, even in the absence of an actual field. If such a field exists, there is no implicit getter for the field. An explicitly declared getter method must take no arguments and return an appropriately typed result. A getter method must not throw any checked exception. Getter names may not overlap ordinary method names. A getter method must be invoked with the field access syntax:

expr.id

where *id* is the name of the getter method.

setter: A method declaration with the modifier `setter` explicitly declares a setter method for a field, even in the absence of an actual field. If such a field exists, there is no implicit setter for the field. An explicitly declared setter method must take a single argument—the value being set—and return (). A setter method must not throw any checked exception. Setter names may not overlap ordinary method names. A setter method must be invoked with the *assignment syntax*:

$expr_1.id := expr_2$

We say that a method declaration *occurs* in a trait declaration. A trait declaration *declares* a method declaration that occurs in that trait declaration. A trait declaration *inherits* method declarations from the declarations of its supertraits. Note that a trait declaration inherits all method declarations declared by all of its supertraits—there’s no real notion of overriding, just overloading (as discussed in Chapter 15). A trait declaration *provides* the method declarations that it declares or inherits.

There are two sorts of method declarations: *dotted method* declarations and *functional method* declarations. Syntactically, a dotted method declaration is identical to a function declaration, except that a special `self` parameter is provided immediately before the name of the method. When a method is invoked, the `self` parameter is bound to the object on which it is invoked. If no `self` parameter is provided explicitly, it is implicitly a parameter with name `self`. An explicit `self` parameter may be an identifier other than `self`, in which case `self` is not necessarily declared within that method.

A functional method declaration does not have a `self` parameter before the method name. Instead, it has a parameter named `self` at an arbitrary position in its parameter list. This parameter is not given a type and implicitly has the type of the enclosing declaration. Semantically, functional method declarations can be viewed as top-level functions. For example, the following overloaded functional method *f* declared within a trait declaration *A*:

```
trait A
  f(self, t: T) = e1
  f(s: S, self) = e2
end
f(a, t)
```

may be rewritten as top-level functions as follows:

```
trait A
  internalF(t: T) = e1
  internalF(s: S) = e2
end
f1(a: A, t: T) = a.internalF(t)
f2(s: S, a: A) = a.internalF(s)
f1(a, t)
```

where *internalF* is a freshly generated name. Functional method declarations may be overloaded with top-level function declarations. An abstract function declaration (described in Section 12.3) can be provided also for overloaded functional method declarations. See Chapter 15 for a discussion of overloaded functionals in Fortress.

A non-`self` `self` parameter can be used within nested object expressions (described in Section 13.9) to name the outer object in methods of the inner:

```
object
  m() = object
```

```

    notSelf.getOuterSelf() = self(* "self" declared in outer scope *)
    getInnerSelf() = self(* regular inner "self" *)
  end
end

```

When a method declaration includes a body expression, it is called a *method definition*. A method declaration that does not have its body expression is referred to as an *abstract method declaration*. An abstract method declaration declares an *abstract method*; any object inheriting an abstract method must define a body expression for the method. An abstract method declaration may include the modifier `abstract`. It may elide parameter names but parameter types cannot be omitted except for the `self` parameter.

Here is an example trait `Enzyme` which provides methods `mass`, `catalyze`, and `reactionSpeed`:

```

trait Enzyme extends { OrganicMolecule, Catalyst }
  reactionSpeed(): Speed
  catalyze(reaction) = reaction.accelerate(reactionSpeed())
end

```

`Enzyme` inherits the abstract method `mass` from `OrganicMolecule`, declares the abstract method `reactionSpeed`, and declares the concrete method `catalyze` which is inherited as an abstract method from its supertrait `Catalyst`.

9.3 Abstract Field Declarations

Syntax:

```

AbsFldDecl ::= AbsFldWTypes
            | AbsFldWoTypes : TypeRef ...
            | AbsFldWoTypes : SimpleTupleType
AbsFldWTypes ::= AbsFldWType
              | ( AbsFldWType(, AbsFldWType)+ )
AbsFldWType ::= AbsFldMods? Id IsType
AbsFldWoTypes ::= AbsFldWoType
               | ( AbsFldWoType(, AbsFldWoType)+ )
AbsFldWoType ::= AbsFldMods? Id
AbsFldMods ::= AbsFldMod+
AbsFldMod ::= hidden | settable | wrapped | UniversalMod

```

Traits may also include abstract field declarations that are implicit declarations of abstract getter methods. Syntactically, an abstract field declaration consists of an optional sequence of modifiers followed by the field name, followed by the token `:`, and the type of the field.

By default, a field declaration implicitly declares a getter method for the field unless there is an explicit getter declared in the enclosing trait. An implicit getter method takes no arguments, has the same name as the field, and has a return type equal to the field type. When called, the implicit getter returns the value of the field when called.

Abstract field declarations can include the following special modifiers:

hidden: A field declaration with the modifier `hidden` has no implicit getter method.

settable: A field declaration with the modifier `settable` has an implicit setter method unless there is an explicit setter declared in the enclosing trait. An implicit setter method takes a parameter (with no default expression) whose type is the type of the field, and returns `()`. When called, the implicit setter rebinds the corresponding field to its

argument. If a field declaration includes the modifier `settable` and `hidden`, only an abstract setter is declared. If a field declaration includes the modifier `hidden` without `settable`, it is a static error.

wrapped: If a field declaration of f has the modifier `wrapped` and the type of f is trait type T , and T is not a naked type variable, then the enclosing trait S implicitly includes “forwarding methods” for all methods in T that are also inherited from any supertrait of S . Each of these methods simply calls the corresponding method on the trait referred to by field f . If the trait declaration enclosing f explicitly declares a method m that conflicts with an implicitly declared forwarding method m' , then the enclosing trait contains only method m , not m' . If the trait declaration enclosing f inherits a concrete method m that conflicts with an implicitly declared forwarding method m' , then the enclosing trait contains only method m , not m' . Because wrapped fields do not change declarations of methods but change definitions of methods, they only affect implementations; APIs do not include wrapped fields.

For example, in the following declarations:

```
trait Dictionary[[T]]
  put(T): ()
  get(): T
end

trait WrappedDictionary[[T]] extends Dictionary[[T]]
  wrapped val: Dictionary[[T]]
  get(): T
end
```

the parametric trait `WrappedDictionary` implicitly includes the following forwarding method:

```
put(x) = val.put(x)
```

If `get` were not explicitly declared in `WrappedDictionary`, then `WrappedDictionary` would also include the forwarding method:

```
get() = val.get()
```

9.4 Method Contracts

Syntax:

```
Contract      ::= Requires? Ensures? Invariant?
Requires      ::= requires { ExprList? }
Ensures       ::= ensures { ProvidedList? }
ProvidedList  ::= Provided( , Provided)*
Provided      ::= Expr (provided Expr)?
Invariant     ::= invariant { ExprList? }
```

Method contracts consist of three optional clauses: a `requires` clause, an `ensures` clause, and an `invariant` clause. All three clauses are evaluated in the scope of the method body. See Section 12.4 for a discussion of each clause.

Method contracts are handled similarly to the manner described in [10]. In particular, substitutability under subtyping is preserved. For a call to a method m with receiver e , we use the term *static contract* of m to refer to a contract declared in the statically most applicable method declaration provided by the static type of e and the term *dynamic contract* of m to refer to a contract declared in the dynamically most applicable method declaration provided by the runtime type of e . Three exceptions may be thrown due to a method contract violation: `CallerViolation` is thrown when the `requires` clause of the static contract fails, `CalleeViolation` is thrown when the `ensures` or `invariant`

clause of the dynamic contract fails, and `ContractHierarchyViolation` is thrown when the `requires` clause of the dynamic contract or the `ensures` or `invariant` clause of the static contract fails.

Evaluation of a call to a method m with receiver e proceeds as follows. First, e is evaluated to a value v with runtime type U . Let C and C' be the static and dynamic contracts of m , respectively. If the `requires` clause of C fails, a `CallerViolation` exception is thrown. Otherwise, if the `requires` clause of C' fails, a `ContractHierarchyViolation` exception is thrown. Otherwise, the `provided` subclauses of C and C' are evaluated. For every `provided` subclause that evaluates to `true`, the corresponding `ensures` subclause is recorded in a table E for later comparison. Similarly, the `invariant` clauses of C and C' are evaluated and the results are stored in E for later comparison. Then the body of m provided by U is evaluated. After evaluation of the body, all `ensures` subclauses of the dynamic contract recorded in E are checked to ensure that they evaluate to `true`, and all `invariant` clauses of the dynamic contract recorded in E are checked to ensure that they evaluate to values equal to the values they evaluated to before evaluation of the body. If any such check fails, a `CalleeViolation` exception is thrown. Otherwise, all `ensures` subclauses and `invariant` clauses of the static contract in E are checked. If any of these checks fails, a `ContractHierarchyViolation` exception is thrown.

9.5 Value Traits

Syntax:

```
TraitMod ::= value
```

If a trait declaration has the modifier `value`, all subtraits of that trait must also have the modifier `value`, and all objects extending that trait are required to be value objects (described in Section 10.3). If a field declaration of a value trait has the modifier `settable`, the return type of its implicit setter method is the value trait type. If a value trait has an explicit setter method, the setter must be an abstract method and its return type must be the value trait type. See Section 10.3 for a discussion of updating fields of value objects.

Chapter 10

Objects

An object is a *value object*, a *reference object*, or a *function object*: It is a function object if it has an arrow type, a reference object if it has an object trait type that is not declared with the `value` modifier (see Section 10.3), and a value object otherwise (i.e., if it has a tuple type, the type `()`, or an object trait type declared with the `value` modifier).

A value object cannot have mutable fields, and it is completely determined by its type, environment and its fields: Value objects with the same type, environment and fields are indistinguishable. Thus, an implementation may freely copy value objects. Most objects with simple standard types, such as booleans, numeric literals, IEEE floating-point numbers, and integers are value objects. In contrast, reference objects are thought to “reside in memory”, and are identified by an *object reference*. A new object reference is created whenever a reference object is constructed, so that reference objects constructed separately are always distinct. Reference objects include arbitrary-precision numbers and aggregates such as arrays, lists and sets. Function objects are immutable and have no fields. Identity is not well-defined for function objects, and attempting to check whether two functions are equivalent returns an approximate result. Section 10.4 describes object equivalence in further detail.

10.1 Object Declarations

Syntax:

```
ObjectDecl ::= ObjectHeader GoInAnObject? end
ObjectHeader ::= ObjectMods? object Id StaticParams? ObjectValParam? Extends? FnClauses
ObjectMods ::= TraitMods
ObjectValParam ::= ( ObjectParams? )
ObjectParams ::= (ObjectParam ,)* (ObjectVarargs ,)? ObjectKeyword( , ObjectKeyword)*
                | (ObjectParam ,)* ObjectVarargs
                | ObjectParam ( , ObjectParam)*
ObjectVarargs ::= transient Varargs
ObjectKeyword ::= ObjectParam = Expr
ObjectParam ::= FldMods? Param
                | transient Param
FldMods ::= FldMod+
FldMod ::= var | AbsFldMod
AbsFldMod ::= hidden | settable | wrapped | UniversalMod
GoInAnObject ::= GoFrontInAnObject GoBackInAnObject?
                | GoBackInAnObject
```

```

GoFrontInAnObject ::= GoesFrontInAnObject+
GoesFrontInAnObject ::= FldDecl
                       | GetterSetterDef
                       | PropertyDecl
GoBackInAnObject ::= GoesBackInAnObject+
GoesBackInAnObject ::= MdDef
                       | PropertyDecl
FnClauses ::= Throws? Where? Contract
Throws ::= throws MayTraitTypes

```

Object declarations declare both object values and object trait types. Object declarations extend a set of traits from which they inherit methods. An object declaration inherits the concrete methods of its supertraits and must include a definition for every method declared but not defined by its supertraits. Especially, an object declaration must not include abstract methods (discussed in Section 9.2); it must define all abstract methods inherited from its supertraits. It is also allowed to define overloaded declarations of concrete methods inherited from its supertraits.

Syntactically, an object declaration begins with an optional sequence of modifiers followed by `object`, followed by the identifier of the object, optional static parameters (described in Chapter 11), optional value parameters, optional traits the object extends, an optional declaration of thrown checked exceptions (discussed in Chapter 14), an optional `where` clause (discussed in Section 11.6), a contract for the object (discussed in Section 12.4), zero or more declarations of fields, getter methods, and setter methods, and zero or more declarations of methods that are not getter or setter, and finally `end`. Property declarations (described in Section 19.6) can be freely commingled with the field and method declarations. Method declarations in object declarations are syntactically identical to method declarations in trait declarations. If an object declaration has no `extends` clause, the object implicitly extends only trait `Object`. A `throws` clause does not include naked type variables. Every element in a `throws` clause is a subtype of `CheckedException`. A contract of an object declaration is evaluated when the object is created, as with a function contract (see Section 12.4).

There are two kinds of object declarations: singleton declarations and constructor declarations. A singleton declaration declares a sole, stand-alone, singleton object. It may have static parameters but it does not have a list of value parameters; every instantiation of such an object with the same static arguments yields the same singleton object. A constructor declaration declares an object constructor. It may have static parameters and it includes a list of value parameters; every call to the constructor of such an object with the same argument yields a new object. Initialization of singleton objects is nondeterministic as described in Chapter 4.

A constructor declaration has a (possibly empty) parenthesized list of value parameters. Each value parameter of a constructor declaration may be preceded by the special modifier `transient` or by a sequence of field modifiers. A value parameter preceded by the modifier `transient` does not correspond to a field in an instantiation of the object: `transient` parameters are not in scope in the object's method declarations, but are in scope when the initial-value expressions of the fields are evaluated during object creation.

For example, the following empty list object extending trait `List`:

```

object Empty extends List
  first() = throw Error
  rest() = throw Error
  cons(x) = Cons(x, self)
  append(xs) = xs
end

```

has no fields and four methods.

Here is an example of a `Cons` object constructor extending trait `List` $[[T]]$:

```

object Cons[[T]](first : T, rest : List[[T]])
  extends List[[T]]

```

```

    cons(x) = Cons(x, self)
    append(xs) = Cons(first, rest.append(xs))
end

```

Note that this declaration implicitly introduces the “factory” function `Cons[[T]](first : T, rest : List[[T]]) : Cons[[T]]` which is used in the body of the object declaration to define the `cons` and `append` methods. Multiple factory functions can be defined by overloading a object constructor with functions. For example: `Cons(first : T) = Cons(first, Empty)`.

10.2 Field Declarations

Syntax:

```

FldDecl ::= FldWTypes InitVal
         | FldWoTypes = Expr
         | FldWoTypes : TypeRef ... InitVal
         | FldWoTypes : SimpleTupleType InitVal
FldWTypes ::= FldWType
           | ( FldWType(, FldWType)+ )
FldWType ::= FldMods? Id IsType
FldWoTypes ::= FldWoType
           | ( FldWoType(, FldWoType)+ )
FldWoType ::= FldMods? Id

```

Fields are variables local to an object. They must not be referred to outside their enclosing object declarations. Each field is either a non-transient value parameter of a constructor declaration, or it is explicitly defined by a field declaration within the body of a constructor declaration, a singleton declaration, or an object expression. A field declaration in an object declaration is syntactically identical to a top-level variable declaration (described in Section 6.2), with the same meanings attached to the form of variable declarations. Additional modifiers apply to fields, as described in Section 9.3. Unlike with an abstract field in a trait declaration, an object declaration may declare a `hidden` field which is not `settable`.

Each field declaration includes an *initial-value* expression, which specifies the initial value of that field. Field initialization occurs in textual program order: evaluation of each initial-value expression must complete before evaluation of the next initial-value expression, and all previous field names (and the parameters of the constructor, in a constructor declaration) are in scope when evaluating an initial-value expression (see Section 7.2). All fields of an object are initialized before that object is made available for subsequent computation; thus, it is illegal to invoke methods on an object being initialized: `self` is *not* implicitly declared by a field declaration.

Within an object declaration or object expression, a field can be accessed by a “naked” identifier reference (see Section 13.2). Unlike within trait declarations, such a reference does *not* invoke the getter or setter method of that name. To invoke a getter or setter of that name, it is necessary to use dotted field access (see Section 13.3).

10.3 Value Objects

An object declaration with the modifier `value` declares a value object that is called in many languages a *primitive* value. The object trait type declared by a value object implicitly has the modifier `value`.

The fields of a `value` object are immutable; they cannot be changed directly or it is a static error. However, Fortress allows value objects to have settable fields as an abbreviation for constructing a new value object with a different value for one field. If a value object has a setter method or a subscripted assignment operator method (described in Section 34.7), then the return type of the method must be the value object trait type instead of `()`. When such a method is invoked, the receiver must itself be assignable, and the value returned by the method is assigned to the receiver.

For example, here is a value object `Complex` number:

```
value object Complex(settable real : Double, settable imaginary : Double = 0)
  opr +(self, other : Complex) = Complex(real + other.real, imaginary + other.imaginary)
end
```

When a mutable variable `z`:

```
var z : Complex = Complex(0)
```

updates its `imaginary` field, the following syntax:

```
z.imaginary := v
```

can be used as an abbreviation for:

```
z := Complex(z.real, v)
```

So the setter for the field `imaginary` in `Complex` would do the work of constructing and returning `Complex(z.real, v)`, and the assignment:

```
z.imaginary := v
```

would be construed to mean:

```
z := z.imaginary(v)
```

Note that modifying a settable field directly within the value object is not allowed. For example, the following:

```
imaginary := 3
```

within the `Complex` object means:

```
self.imaginary := 3
```

and because `self` is not mutable, the assignment is disallowed.

10.4 Object Equivalence

The trait `Any` declares the object equivalence operator `≡`. This operator is automatically defined for all objects; it is a static error for the programmer to override it. The `≡` operator is used to decide whether its two arguments refer to “the same object” in the strictest sense possible. If the arguments have different dynamic types—including the instantiations of all static parameters—the result is always *false*. If both arguments are value objects with the same type, then the result is *true* if and only if corresponding fields of the objects are themselves equivalent as defined by this operator; in particular, two binary words are strictly equivalent if and only if they contain the same bit pattern. If both arguments are object references, then the result is *true* if and only if the two object references refer to the identical reference object (in implementation terms, occupying the same memory locations in the heap). If both arguments are functions, the result is *true* only if the functions behave identically for any choice of type-correct arguments. Even if two functions behave identically, the fortress implementation is free to return *false* when they are compared for object equivalence.

Chapter 11

Static Parameters

Trait, object, and functional declarations may be parameterized with *static parameters*. Static parameters are *static variables* listed in white square brackets `[[` and `]]` immediately after the name of a trait, object, or functional and they are in scope of the entire body of the declaration. Static parameters may be instantiated with static expressions discussed in Section 13.26. In this chapter, we describe the forms that these static parameters can take.

Syntax:

```
StaticParams ::= [[StaticParamList]]
StaticParamList ::= StaticParam( , StaticParam)*
```

11.1 Type Parameters

Syntax:

```
StaticParam ::= Id Extends? ( absorbs unit)?
```

Static parameters may include one or more type parameters. Syntactically, a type parameter consists of an identifier followed by an optional `extends` clause, followed by an optional “`absorbs unit`” clause (described in Section 35.4). If a type parameter does not have an `extends` clause, it has an implicit “`extends Object`” clause.

Type parameters are instantiated with types such as trait types, tuple types, and arrow types (See Chapter 8 for a discussion of Fortress types). We use the term *naked type variable* to refer to an occurrence of a type variable as a stand-alone type (rather than as a parameter to another type). Type parameters can appear in any context that an ordinary type can appear, except that a naked type variable must not appear in the `extends` clause of a trait or object declaration nor as the type of a `wrapped` field (discussed in Section 10.2).

Here is a parameterized trait `List`:

```
trait List[[T]]
  first(): T
  rest(): List[[T]]
  cons(T): List[[T]]
  append(List[[T]]): List[[T]]
end
```

11.2 Nat and Int Parameters

Syntax:

```
StaticParam ::= nat Id
              | int Id
```

Static parameters may include one or more `nat` and `int` parameters. Syntactically, a `nat` parameter consists of `nat` followed by an identifier. An `int` parameter consists of `int` followed by an identifier. These parameters are instantiated at runtime with numeric values. A `nat` parameter may be used to instantiate other `nat` parameters, or to appear in any context that a variable of type \mathbb{N} can appear, except that it cannot be assigned to. An `int` parameter may be used to instantiate other `int` parameters, or to appear in any context that a variable of type \mathbb{Z} can appear, except that it cannot be assigned to.

For example, the following function f :

```
f[[nat n]](x: Length2n): Lengthn = sqrt(x)
```

declares a `nat` parameter n , which appears in both the parameter type and return type of f .

11.3 Bool Parameters

Syntax:

```
StaticParam ::= bool Id
```

Static parameters may include one or more `bool` parameters. Syntactically, a `bool` parameter consists of `bool` followed by an identifier. These parameters are instantiated at runtime with boolean values. They may be used to instantiate other `bool` parameters, or to appear in any context that a variable of type `Boolean` can appear, except that they cannot be assigned to.

For example, the following `coercion` declared in the trait `Boolean`:

```
trait Boolean
  coercion [[bool b]](x: BooleanLiteral[[b]])
end
```

declares a `bool` parameter b , which appears in the parameter type. See Chapter 24 for a full declaration of `Boolean`.

11.4 Dimension and Unit Parameters

Syntax:

```
StaticParam ::= dim Id
              | unit Id (: DimRef)? (absorbs unit)?
```

Static parameters may include one or more `dim` and `unit` parameters. Syntactically, a `dim` parameter begins with `dim` followed by an identifier. A `unit` parameter begins with `unit` followed by an identifier, optionally followed by the token `:` and a dimension, and the unit is thereby restricted to be a unit of the specified dimension. A `unit` parameter may include the clause `“absorbs unit”`; the meaning of this is described in Section 35.4. A `dim` parameter is allowed to appear in any context that a dimension can appear. A `unit` parameter is allowed to appear in any context that a unit can appear.

For example, here is a function that is parameterized with a unit:

```
sqrt[[unit U]](x: ℝ64 U2): ℝ64 U = numericalsqrt(x/U2) U
```

11.5 Operator and Identifier Parameters

Syntax:

```
StaticParam ::= opr Op
              | ident Id
```

Static parameters may include one or more operator names and identifiers denoting method names. Syntactically, an operator parameter begins with `opr` followed by an operator name. An identifier parameter begins with `ident` followed by an identifier.

Operator and identifier parameters may be freely intermixed with other static parameters. For example, the following trait `HasLeftZeroes`:

```
trait HasLeftZeroes[T extends HasLeftZeroes[T, ⊙, isLeftZero], opr ⊙, ident isLeftZero]
  extends { BinaryOperator[T, ⊙] }
  abstract isLeftZero(): Boolean
  property ∀(a: T, b: T) a.isLeftZero() →: ((a ⊙ b) = a)
end
```

is parameterized with a type parameter T , an operator parameter \odot , and an identifier parameter `isLeftZero`.

Unlike other static parameters, operator and identifier parameters may be used in both type context and value context. The following example operator parameter \odot :

```
trait IdentityOp[T extends IdentityOp[T, ⊙], opr ⊙]
  opr ⊙(self): T = self
end
```

is declared as the second static parameter of `IdentityOp`, instantiated as a static argument in `IdentityOp[T, ⊙]` which is the bound of T , and declared as an operator method in `IdentityOp`.

Operator parameters and identifier parameters are instantiated with operators and identifiers respectively. They may be used to instantiate other operator and identifier parameters and the names of method declarations. For example, the following trait `MyIdentity`:

```
object MyIdentity extends IdentityOp[MyIdentity, IDENTITY] end
```

instantiates the operator parameter of its supertrait `IdentityOp` with the operator name `IDENTITY`. It inherits the `IDENTITY` operator from `IdentityOp`. One restriction applies to instantiations of operator and identifier parameters: one operator (or identifier) name cannot instantiate two or more operator (or identifier) parameters of a single trait or object.

An operator method declaration whose name is one of the operator parameters of its enclosing trait or object may be overloaded with other operator declarations in the same component; the operator parameter may be instantiated with any operator in the same component. Therefore, such an operator method declaration must satisfy the overloading rules (described in Chapter 33) with every operator declaration in the same component. For example, the above `IDENTITY` operator declaration is overloaded with the following `IDENTITY` operator declaration:

```
opr IDENTITY(x: ℤ) = x
opr +(x: ℤ) = x
```

where both `IDENTITY` and `+` are top-level operator declarations in the same component. Therefore, the declaration of the operator \odot in `IdentityOp` must satisfy the overloading rules with `IDENTITY` and with `+`.

Many interesting examples are described in Section 37.3.

11.6 Where Clauses

Syntax:

```
Where ::= where { WhereClauseList }
WhereClauseList ::= WhereClause(, WhereClause)*
WhereClause ::= Id Extends
                | TypeAlias
                | NatConstraint
                | IntConstraint
                | BoolConstraint
                | UnitConstraint
                | TypeRef coerces TypeRef
                | TypeRef widens TypeRef
```

Static parameters may have constraints placed on them in a `where` clause. A `where` clause begins with `where`, followed by a sequence of static parameter constraints enclosed in braces `{` and `}`, and separated by commas.

A `where` clause may introduce new static variables, i.e., identifiers for types and other static entities that may not be static parameters. We use the term *where-clause variables* to refer to static variables that are not also static parameters. The `where`-clause variables must be bound in a `where` clause.

A static parameter constraint is one of the following forms:

- a trait constraint consisting of the identifier of a naked type variable, followed by `extends` followed by a set of trait references which may include naked type variables,
- a type alias (described in Section 8.9),
- an arithmetic constraint,
- a boolean constraint,
- a unit equality constraint,
- a `coerces` constraint (described in Section 17.2), or
- a `widens` constraint (described in Section 17.2).

A `where` clause may include mutually recursive constraints. All static variables in a trait, object, or functional declaration must occur either as a static parameter or as a `where`-clause variable. Appendix A.2 describes a Fortress core calculus with `where` clauses.

Trait declarations are allowed to extend other instantiations of themselves. For example, we can write:

```
trait C[[S]] extends C[[T]]
  where {S extends T, T extends Object}
end
```

In this declaration, for every subtype S of T , $C[[S]]$ is a subtype of $C[[T]]$. Effectively, we have expressed the fact that the static parameter S of C is covariant.

Trait declarations need not have any static parameters in order to have a `where` clause. For example, the following trait declaration is legal:

```
trait C extends D[[T]]
  where {T extends Object}
end
```

In this declaration, trait C is a subtrait of *every* instantiation of parametric trait D . Thus, trait C has all of the methods of every instantiation of D . By thinking of the declaration this way, we can see what restrictions we need to impose on the trait C in order for it to be sensible. If trait C inherits a method declaration that refers to T , it really contains infinitely many methods (one for each instantiation of T). However, instantiations of the *where*-clause variables are not explicit from the program text as static parameters are. It must be possible to infer which method is referred to at the call site. If there is not enough information to infer which method is called, type checking rejects the program and requires more type information from the programmer. Programmers always can provide more type information by using type ascription as described in Section 13.29.

Object or functional declarations may include *where* clauses. Here is an example declaration of an Empty list:

```
object Empty extends List[[T]] where {T extends Object}
  first() = throw Error
  rest() = throw Error
  cons(x) = Cons(x, self)
  append(xs) = xs
end
```

where `Cons` is declared in Section 10.1.

Chapter 12

Functions

Functions are values that have arrow types described in Section 8.5. Each function takes exactly one argument, which may be a tuple, and returns exactly one result, which may be a tuple. A function may be declared as top level or local as described in Section 6.1. Fortress allows functions to be *overloaded* (as described in Chapter 15); there may be multiple function declarations with the same function name in a single lexical scope. Functions can be passed as arguments and returned as values. Single variables may be bound to functions including overloaded functions.

12.1 Function Declarations

Syntax:

```
FnDecl ::= FnDef
          | AbsFnDecl
FnDef ::= FnMods? FnHeaderFront FnHeaderClause = Expr
FnMods ::= FnMod+
FnMod ::= atomic | io | UniversalMod
FnHeaderFront ::= Id StaticParams? ValParam
                  | OpHeaderFront
ValParam ::= BindId
              | (Params?)
BindId ::= Id
            | -
Params ::= (Param ,)* (Varargs ,)? Keyword( , Keyword)*
            | (Param ,)* Varargs
            | Param( , Param)*
VarargsParam ::= Id : TypeRef ...
Varargs ::= VarargsParam
Keyword ::= Param = Expr
PlainParam ::= BindId IsType?
                | TypeRef
Param ::= PlainParam
FnHeaderClause ::= IsType? FnClauses
FnClauses ::= Throws? Where? Contract
Throws ::= throws MayTraitTypes
```

Syntactically, a function declaration consists of an optional sequence of modifiers followed by the name of the function, optional static parameters (described in Chapter 11), the value parameter with its (optionally) declared type, an optional type of a return value, an optional declaration of thrown checked exceptions (discussed in Chapter 14), an optional `where` clause (discussed in Section 11.6), a contract for the function (discussed in Section 12.4), and finally an optional body expression preceded by the token `=`. A `throws` clause does not include naked type variables. Every element in a `throws` clause is a subtype of `CheckedException`. When a function declaration includes a body expression, it is called a *function definition*. Function declarations can be mutually recursive.

Function declarations can include the following special modifiers:

atomic: A function with the modifier `atomic` acts as if its entire body were surrounded in an `atomic` expression discussed in Section 13.22.

io: Functions that perform externally visible input/output actions are said to be `io` functions. An `io` function must not be invoked from a non-`io` function.

A function takes exactly one argument, which may be a tuple. When a function takes a tuple argument, we abuse terminology by saying that the function takes multiple arguments. Value parameters cannot be mutated inside the function body.

A function's value parameter consists of a parenthesized, comma-separated list of bindings where each binding is one of:

- A plain binding “*identifier*” or “*identifier* : *T*”
- A varargs binding “*identifier* : *T* . . .”
- A keyword binding “*identifier* = *e*” or “*identifier* : *T* = *e*”

When the parameter is a single plain binding without a declared type, enclosing parentheses may be elided. The following restrictions apply: No two bindings may have the same identifier. No keyword binding may precede a plain binding. No varargs binding may follow a keyword binding or precede a plain binding. Note that it is permitted to have a single plain binding, or to have no bindings. The latter case, “()”, is considered equivalent to a single plain binding of the ignored identifier “_” of type `()`, that is, “(- : ())”. Also, there can be at most one varargs binding.

A parameter declared by keyword binding is called a *keyword parameter*; a keyword parameter must be declared with a *default* expression, which is used when no argument is bound to the parameter explicitly. Syntactically, the default expression is specified after an `=` sign. The default expression of a parameter *x* of function *f* is evaluated each time the function is called without a value provided for *x* at the call site. All parameters occurring to the left of *x* are in scope of its default expression. All parameters following *x* must include default expressions as well; *x* is in scope of their default expressions and the body of the function. When an argument is passed explicitly for a keyword parameter, that argument must be passed as a *keyword argument*. (See Section 12.2.) If no type is declared for a keyword parameter, the type is inferred from the static type of its default expression.

A parameter declared by varargs binding is called a *varargs parameter*; it is used to pass a variable number of arguments to a function as a single heap sequence. The type of a varargs parameter is `HeapSequence[[T]]` where *T* is the type mentioned in (or inferred for) that binding. See Section 40.3 for a discussion of `HeapSequence`. Note that the type of a varargs parameter cannot be omitted. If a function does not have a varargs parameter then the number of arguments is fixed by the function's type. Note that a varargs parameter is not allowed to have a default expression.

For example, here is a simple polymorphic function for creating lists:

```
List[[T extends Object, nat length]](rest : T[length]) =
  if length = 0 then Empty
  else Cons(rest0, List(rest1:(length-1)))
end
```

The following function:

$$\text{swap}(x : \text{Object}, y : \text{Object}) : (\text{Object}, \text{Object}) = (y, x)$$

has no static parameters, throws no checked exceptions, and has no contract. It takes a tuple of two elements of type `Object` and returns a tuple of two values. Namely, it returns its arguments in reverse order. If the return type is elided, it is inferred to be the static type of the body. The following declaration of `swap` has the same return type as the above declaration:

$$\text{swap}(x : \text{Object}, y : \text{Object}) = (y, x)$$

Similarly, function parameter type can often be inferred from the body of the function. When a type can be inferred for a parameter from the body of the function, that parameter type need not be declared explicitly. Thus, the following declaration of `swap` has the same parameter type and return type as the above declarations:

$$\text{swap}(x, y) = (y, x)$$

See Chapter 20 for a discussion of type inference in Fortress.

The following function `wrap`:

$$\text{wrap}(xs, ys = xs) = [xs\ ys]$$

returns an array containing its parameters. If a value for only the parameter `xs` is given to `wrap` at a call site, the value of `xs` is bound to `ys` as well, and an array that contains `xs` as both of its indices is returned.

12.2 Function Applications

Fortress provides overloaded functions (as described in Chapter 15); there may be multiple function declarations with the same function name in a single lexical scope. Thus, we need to determine which function declarations are applicable to a function application.

If a function's argument type is `()`, then function declarations with the following forms of parameter lists are considered to be applicable:

- `()` which means the same thing as `(_: ())`
- `(x : ())` which is something programmers don't ordinarily write
- `(x : T ...)`

In the last case, `x` is bound to an empty `HeapSequence[[T]]`.

If a function's argument type `A` is neither `()` nor a tuple type, then function declarations with the following forms of parameter lists are considered to be applicable:

- `(x : T)` where `A` is a subtype of `T`
- `(x : T ...)` where `A` is a subtype of `T`

In the last case, `x` is bound to a `HeapSequence[[T]]` of length 1, containing the actual argument value.

If a function's argument type `A` is a tuple type, then function declarations with the following forms of parameter lists are considered to be applicable:

- `(x : T)` where `A` is a subtype of `T`
- `(x : T ...)` where `A` is a subtype of `T`
- a parameter list with no varargs binding, provided that

- type A has exactly as many plain types as the parameter list has plain bindings, and
 - for every keyword-type pair (described in Section 8.4) in A , the parameter list has a binding with the same keyword, and
 - for every element type in A , the type in the element type is a subtype of the type of the corresponding binding in the parameter list.
- a parameter list with a varargs binding, provided that
 - type A has at least as many plain types as the parameter list has plain bindings, and
 - for every keyword-type pair in A , the parameter list has a binding with the same keyword, and
 - for every element type in A , the type in the element type is a subtype of the type of the corresponding binding in the parameter list—but if there is no corresponding binding, then the type in the element type must be a subtype of the type in the varargs binding.

In the latter case, the parameter named by the identifier in the varargs binding is bound to a `HeapSequence[[T]]` that contains, in order, all the values of the tuple that did not correspond to plain bindings, followed by all the values in the varargs `HeapSequence` of the tuple, if any.

When an argument is passed explicitly for a keyword parameter, that argument must be passed as a *keyword argument*. Syntactically, a keyword argument is a keyword-value pair “*identifier = e*”. Keyword parameters not explicitly bound are bound to their default values. If a parameter that has no default value is not explicitly bound to an argument, it is a static error. Because a keyword-value pair shares a syntax with an *equality expression*, we provide rules for disambiguation in Section 13.27.6.

When a function is called (See Section 13.6 for a discussion of function call expressions), explicit arguments are evaluated in parallel, keyword parameters not explicitly bound are bound to their default values sequentially, and the body of the function is evaluated in a new environment, extending the environment in which it is defined with all parameters bound to their arguments.

If the application of a function f ends by calling another function g , tail-call optimization must be applied. Storage used by the new environments constructed for the application of f must be reclaimed.

Here are some examples:

```
sqrt(x)
arctan(y, x)
makeColor(red = 5, green = 3, blue = 43)
processString(s, start = 5, finish = 43)
```

If the function’s argument is not a tuple, then the argument need not be parenthesized:

```
sqrt 2
sin x
log log n
```

Here are a few varargs and keyword examples:

```
f(x : ℤ, y : ℤ . . . , z : ℤ = 0) = ⟨x, ⟨q | q ← y⟩, z⟩

f(1)           returns           ⟨1, ⟨⟩, 0⟩
f(1, 2, 3)     returns           ⟨1, ⟨2, 3⟩, 0⟩
f(1, [2 3] . . .) returns           ⟨1, ⟨2, 3⟩, 0⟩
f(1, 2, 3, [4 5] . . .) returns       ⟨1, ⟨2, 3, 4, 5⟩, 0⟩
f(1, 2, 3, 17 # 3 . . .) returns       ⟨1, ⟨2, 3, 17, 18, 19⟩, 0⟩
f(1, 2, 3, z = 8) returns             ⟨1, ⟨2, 3⟩, 8⟩
f([2 3] . . .) declaration not applicable
```

12.3 Abstract Function Declarations

Syntax:

```
AbsFnDecl ::= FnMods? FnHeaderFront FnHeaderClause  
           | Name : ArrowType
```

A function with a single principal type may be declared separately from its definitions using an *abstract function declaration*. It is a static error if there are two abstract function declarations with the same name. For an abstract function declaration with name f , argument type T , and return type U (T and U may be tuple types), any concrete declaration for f must be for a function whose argument type is a subtype of T and whose return type is a subtype of U . Furthermore, the union of the argument types of the concrete declarations for f must be equal to T . Concrete function declarations must satisfy the overloading rules. Unless T has a `comprises` clause (or is a tuple type, at least one of whose entries has a `comprises` clause), this implies that some concrete declaration for f must have argument type T .

Syntactically, an abstract function declaration is a function declaration without a body. Parameter names may be elided but parameter types cannot be omitted. Additionally, when a function's type is not parameterized, Fortress provides an alternative mathematical notation for an abstract function declaration: function name followed by the token `' : '`, followed by an arrow type.

For example, after the following abstract function declaration:

```
printMolecule(Molecule): ()
```

where trait `Molecule` is defined as follows:

```
trait Molecule comprises {OrganicMolecule, InorganicMolecule} end
```

the programmer could write:

```
printMolecule(molecule: Molecule) = ...
```

or could write:

```
printMolecule(molecule: OrganicMolecule) = ...  
printMolecule(molecule: InorganicMolecule) = ...
```

For the latter, the programmer must provide a definition for every immediate subtype of `Molecule`, or it is a static error.

12.4 Function Contracts

Syntax:

```
Contract ::= Requires? Ensures? Invariant?  
Requires ::= requires { ExprList? }  
Ensures ::= ensures { ProvidedList? }  
ProvidedList ::= Provided(, Provided)*  
Provided ::= Expr(provided Expr)?  
Invariant ::= invariant { ExprList? }
```

Function contracts consist of three optional clauses: a `requires` clause, an `ensures` clause, and an `invariant` clause. All three clauses are evaluated in the scope of the function body.

The `requires` clause consists of a sequence of expressions of type `Boolean` separated by commas and enclosed in curly braces. The `requires` clause is evaluated during a function call before the body of the function. If any expression in a `requires` clause does not evaluate to `true`, a `CallerViolation` exception is thrown.

The `ensures` clause consists of a sequence of `ensures` subclauses. Each such subclause consists of an expression of type `Boolean`, optionally followed by a `provided` subclause. A `provided` subclause begins with `provided` followed by an expression of type `Boolean`. For each subclause in the `ensures` clause of a contract, the `provided` subclause is evaluated immediately after the `requires` clause during a function call (before the function body is evaluated). If a `provided` subclause evaluates to `true`, then the expression preceding this `provided` subclause is evaluated after the function body is evaluated. If the expression evaluated after function evaluation does not evaluate to `true`, a `CalleeViolation` exception is thrown. The expression preceding the `provided` subclause can refer to the return value of the function. A `result` variable is implicitly bound to a return value of the function and is in scope of the expression preceding the `provided` subclause. The implicitly declared `result` shadows any other declaration with the same name in scope.

The `invariant` clause consists of a sequence of expressions of *any type* enclosed by curly braces. These expressions are evaluated before and after a function call. For each expression e in this sequence, if the value of e when evaluated before the function call is not equal to the value of e after the function call, a `CalleeViolation` exception is thrown.

Here are some examples:

```
factorial(n: Z64) requires { n ≥ 0 } =
  if n = 0 then 1
  else n factorial(n - 1)
end
```

```
mangle(input: List) ensures { sorted(result) provided sorted(input) } =
  if input ≠ Empty
  then mangle(first(input))
       mangle(rest(input))
  end
```

Overloaded function contracts are handled similarly with method contracts described in Section 9.4. In particular, substitutability is preserved: the statically most applicable function to a call should be substitutable with the dynamically most applicable function to the call. For a call of function f , we use the term *static contract* of f to refer to a contract declared in the statically most applicable function declaration and the term *dynamic contract* of f to refer to a contract declared in the dynamically most applicable function declaration. Three exceptions may be thrown due to an overloaded function contract violation: `CallerViolation` is thrown when the `requires` clause of the static contract fails, `CalleeViolation` is thrown when the `ensures` or `invariant` clause of the dynamic contract fails, and `ContractOverloadingViolation` is thrown when the `requires` clause of the dynamic contract or the `ensures` or `invariant` clause of the static contract fails.

Evaluation of a call of function f proceeds as follows. Let C and C' be the static and dynamic contracts of f , respectively. If the `requires` clause of C fails, a `CallerViolation` exception is thrown. Otherwise, if the `requires` clause of C' fails, a `ContractOverloadingViolation` exception is thrown. Otherwise, the `provided` subclauses of C and C' are evaluated. For every `provided` subclause that evaluates to `true`, the corresponding `ensures` subclause is recorded in a table E for later comparison. Similarly, the `invariant` clauses of C and C' are evaluated and the results are stored in E for later comparison. Then the body of the dynamically most applicable function declaration of f is evaluated. After evaluation of the body, all `ensures` subclauses of the dynamic contract recorded in E are checked to ensure that they evaluate to `true`, and all `invariant` clauses of the dynamic contract recorded in E are checked to ensure that they evaluate to values equal to the values they evaluated to before evaluation of the body. If any such check fails, a `CalleeViolation` exception is thrown. Otherwise, all `ensures` subclauses and `invariant` clauses of the static contract in E are checked. If any of these checks fails, a `ContractOverloadingViolation` exception is thrown.

Chapter 13

Expressions

Fortress is an expression-oriented language. The positions in which an expression may legally appear (*value context*) are determined by the nonterminal *Expr* in the Fortress grammar defined in Appendix G. We say that an expression is a *subexpression* of any expression (or any other program construct) that (syntactically) contains it. When evaluation of one subexpression must complete before another subexpression is evaluated, those subexpressions are ordered by dynamic program order (see Chapter 4). This constrains the memory behavior of program constructs, as described in Chapter 21. Unless otherwise specified, abrupt completion of the evaluation of a subexpression causes the evaluation of the expression as a whole to complete abruptly in the same way. Also, if one expression precedes another by dynamic program order, and the evaluation of the first expression completes abruptly, the second is not evaluated at all.

13.1 Literals

Syntax:

```
Literal ::= ( )  
          | NumericLiteral  
          | CharLiteral  
          | StringLiteral
```

Fortress provides boolean literals, `()` literal, character literals, string literals, and numeric literals. Literals are values; they do not require evaluation.

The literal *false* has type `BooleanLiteral[false]`. The literal *true* has type `BooleanLiteral[true]`.

The literal `()` is the only value with type `()`. Whether any given occurrence of `()` refers to the value `()` or to the type `()` is determined by context.

A character literal has type `Character`. Each character literal consists of an abstract character in Unicode 5.0 [27], enclosed in single quotation marks (for example, `'a'`, `'A'`, `'$'`, `'α'`, `'⊕'`). For convenience, the single quotes may be either true typographical “curly” single quotation marks or a pair of ordinary apostrophe characters (for example, `'a'`, `'A'`, `'$'`, `'α'`, `'⊕'`). See Section 5.9 for a description of how names of characters may be used rather than actual characters within character literals, for example `'APOSTROPHE'` and `'GREEK_CAPITAL_LETTER_GAMMA'`.

A string literal has type `String`. Each string literal is a sequence of Unicode 5.0 characters enclosed in double quotation marks (for example, `"Hello, world!"` or `"π r2"`). For convenience, the double quotes may be either true typographical “curly” double quotation marks or a pair of “neutral” double-quote characters (for example, `"Hello, world!"` or `"π r2"`). Section 5.10 also describes how names of characters may be used rather than actual characters within string literals. One may also use the escape sequences `\b` and `\t` and `\n` and `\f` and `\r` as described in [5].

Numeric literals in Fortress are referred to as *numerals*, corresponding to various expressible numbers. Numerals may be either *simple* or *compound* (as described in Section 5.13).

A numeral containing only digits (let n be the number of digits) has type `NaturalNumeral[[n , 10, v]]` where v is the value of the numeral interpreted in radix ten. If the numeral has no leading zeros, or is the literal 0, then it also has type `Literal[[v]]`.

A numeral containing only digits (let n be the number of digits) then an underscore, then a radix indicator (let r be the radix) has type `NaturalNumeralWithExplicitRadix[[n , r , v]]` where v is the value of the n -digit numeral interpreted in radix r .

A numeral containing only digits (let n be the total number of digits) and a radix point (let m be the number of digits after the radix point) has type `RadixPointNumeral[[n , m , 10, v]]` where v is the value of the numeral, with the radix point deleted, interpreted in radix ten.

A numeral containing only digits (let n be the total number of digits) and a radix point (let m be the number of digits after the radix point), then an underscore, then a radix indicator (let r be the radix) has the following type:

`RadixPointNumeralWithExplicitRadix[[n , m , r , v]]`

where v is the value of the n -digit numeral, with the radix point deleted, interpreted in radix r .

Every numeral also has type `Numeral[[n , m , r , v]]` for appropriate values of n , m , r , and v .

Numerals are not directly converted to any of the number types because, as in common mathematical usage, we expect them to be polymorphic. For example, consider the numeral 3.1415926535897932384; converting it immediately to a floating-point number may lose precision. If that numeral is used in an expression involving floating-point intervals, it would be better to convert it directly to an interval. Therefore, numerals have their own types as described above. This approach allows library designers to decide how numerals should interact with other types of objects by defining coercion operations (see Section 17.1 for an explanation of coercion in Fortress). The Fortress standard libraries define coercions from numerals to integers (for simple numerals) and rational numbers (for compound numerals).

In Fortress, dividing two integers using the `/` operator produces a *rational number*; this is true regardless of whether the integers are of type `Z` (or `ZZ`), `Z64` (or `ZZ64`), `N32` (or `NN32`), or whatever. Addition, subtraction, multiplication, and division of rationals are always exact; thus values such as $1/3$ are represented exactly in Fortress.

Numerals containing a radix point are actually rational literals; thus 3.125 has the rational value $3125/1000$. The quotient of two integer literals is a static expression (described in Section 13.26) whose value is rational. Similarly, a sequence of digits with a radix point followed by the symbol \times and an integer literal raised to an integer literal, such as 6.0221415×10^{23} is a static expression whose value is rational. If such static expressions are mentioned as part of a floating-point computation, the compiler performs the rational arithmetic exactly and then converts the result to a floating-point value, thus incurring at most one floating-point rounding error. But in general rational computations may also be performed at run time, not just at compile time.

A rational number can be thought of as a pair of integers p and q that have been reduced to “standard form in lowest terms”; that is, $q > 0$ and there is no nonzero integer k such that $\frac{p}{k}$ and $\frac{q}{k}$ are integers and $|\frac{p}{k}| + |\frac{q}{k}| < |p| + |q|$. The type `Q` includes all such rational numbers.

The type `Q*` relaxes the requirement $q > 0$ to $q \geq 0$ and includes two extra values, $1/0$ and $-1/0$ (sometimes called “the infinite rational” and “the indefinite rational”). The advantage of `Q*` is that it is closed under the rational operations $+$, $-$, \times , and $/$. If a value of type `Q*` is assigned to a variable of type `Q`, a `DivideByZeroException` is thrown at run time if the value is $1/0$ or $-1/0$. The type `Q#` includes all of $1/0$, $-1/0$, and $0/0$. In ASCII, `Q`, `Q*`, and `Q#` are written as `QQ`, `QQ_star`, and `QQ_splat`, respectively. See Section 38.1 for definitions of `Q`, `Q*`, and `Q#`.

13.1.1 Pi

The object named π (or `pi`) may be used to represent the ratio of the circumference of a circle to its diameter rather than a specific floating-point value or interval value. In Fortress, π has type `RationalValueTimesPi[[false, 1, 1]]`. When used in a floating-point computation, it becomes a floating-point value of the appropriate precision; when used in an interval computation, it becomes an interval of the appropriate precision.

13.1.2 Infinity and Zero

The object named ∞ has type `ExtendedIntegerValue[[true, 0, true]]`. One can negate ∞ to get a negative infinity.

Negating the literal `0` produces a special negative-zero object, which refuses to participate in compile-time arithmetic (discussed in Section 13.26). It has type `NegativeZero`. The main thing it is good for is coercion to a floating-point number (discussed in Chapter 17). (Negating any other zero-valued expression simply produces zero.)

13.2 Identifier References

Syntax:

```
Primary ::= Id[[StaticArgList ]
BaseExpr ::= Id
          | self
```

A name that is not an operator appearing in an expression context is called an *identifier reference*. It evaluates to the value of the name in the enclosing scope in the value namespace. The type of an identifier reference is the declared type of the name. See Chapter 7 for a discussion of names. An identifier reference performs a memory read operation. Note in particular that if a name is not in scope, it is a static error (as described in Section 7.2).

An identifier reference which denotes a polymorphic function may include explicit static arguments (described in Chapter 12) but most identifier references do not include them; the static arguments are statically inferred from the context of the method invocation (as described in Chapter 20). For example, `identity[[String]]` is an identifier reference with an explicit static argument where the function `identity` is defined as follows:

```
identity[[T extends Any]](x: T): T = x
```

The special name `self` is declared as a parameter of a method. When the method is invoked, its receiver is bound to the `self` parameter; the value of `self` is the receiver. The type of `self` is the type of the trait or object being declared by the innermost enclosing trait or object declaration or object expression. See Section 9.2 for details about `self` parameters.

13.3 Dotted Field Accesses

Syntax:

```
Primary ::= Primary . Id
```

An expression consisting of a single subexpression (called the *receiver expression*), followed by `.`, followed by an identifier, not immediately followed by a parenthesis, is a *field access*. If the receiver expression denotes an object (called the *receiver*), the field access is evaluated to a call to a getter mapped from that identifier by the receiver. The type of the field access is the return type of its getter. The static type of the receiver indicates whether a getter mapped from that identifier is provided by the denoted object. If a getter is not provided, it is a static error. See Section 9.2

for a discussion of getters. The evaluation of the receiver must complete normally before the body of the getter is evaluated.

13.4 Dotted Method Invocations

Syntax:

```
Primary ::= Primary . Id ([[StaticArgList]])? TupleExpr
         | Primary . Id ([[StaticArgList]])? ()
         | TraitType . coercion ([[StaticArgList]])? (Expr)
```

A *dotted method invocation* consists of a subexpression (called the receiver expression), followed by ‘.’, followed by an identifier, an optional list of static arguments (described in Chapter 12) and a subexpression (called the *argument expression*). Unlike in function calls (described in Section 13.6), the argument expression must be parenthesized, even if it is not a tuple. There must be no whitespace on either side of the ‘.’, and there must be no whitespace on the left-hand side of the left parenthesis of the argument expression. The receiver expression evaluates to the receiver of the invocation (bound to the self parameter (discussed in Section 9.2) of the method). A *coercion* invocation (discussed in Chapter 17) has a similar syntax to a dotted method invocation. A method invocation may include explicit instantiations of static parameters but most method invocations do not include them; the static arguments are statically inferred from the context of the method invocation (as described in Chapter 20).

The receiver and arguments of a method invocation are each evaluated in parallel in a separate implicit thread (see Section 4.4). After this thread group completes normally, the body of the method is evaluated with the parameter of the method bound to the value of the argument expression (thus evaluation of the body occurs after evaluation of the receiver and arguments in dynamic program order). The value and the type of a dotted method invocation are the value and the type of the method body.

We say that methods or functions (collectively called as *functionals*) may be *applied to* (also “*invoked on*” or “*called with*”) an argument. We use “call”, “invocation”, and “application” interchangeably.

Here are some examples:

```
myString.toUpperCase()
myString.replace("foo", "few")
SolarSystem.variation(( $\pi/2$  radian)/452 million year)
myNum.add(otherNum) (* NOT myNum.add otherNum *)
```

13.5 Naked Method Invocations

Syntax:

```
Primary ::= Id TupleExpr
         | Id ()
         | Id Primary
```

Method invocations that are not prefixed by receivers are *naked method invocations*. A naked method invocation is either a functional method call (See Section 9.2 for a discussion of functional methods) or a method invocation within a trait or object that provides the method declaration. Syntactically, a naked method invocation is same as a function call except that the method name is used instead of an arbitrary expression denoting the applied method. Like function calls, an argument expression need not be parenthesized unless it is a tuple. After the evaluation of the argument expression completes normally, the body of the method is evaluated with the parameter of the method bound to the value of the argument expression. The value and the type of a naked method invocation are the value and the type of the method body.

13.6 Function Calls

Syntax:

```
Primary ::= Primary TupleExpr
         | Primary ()
         | Primary Primary
```

A *function call* consists of two subexpressions: an expression denoting the applied function and an argument expression. The argument expression and the expression denoting the applied function are evaluated *in parallel* in separate implicit threads (see Section 4.4). As with languages such as Scheme and the Java Programming Language, function calls in Fortress are call-by-value. After the evaluation of the function and its arguments completes normally, the body of the function is evaluated with the parameter of the function bound to the value of the argument expression. The value and the type of a function call are the value and the type of the function body.

Here are some examples:

```
sqrt(x)
arctan(y, x)
```

If the function's argument is not a tuple, then the argument need not be parenthesized:

```
sqrt 2
sin x
log log n
```

13.7 Function Expressions

Syntax:

```
Expr ::= fn ValParam IsType? Throws? => Expr
```

Function expressions denote function values; they do not require evaluation. Syntactically, they start with `fn` followed by a parameter, optional return type, optional `throws` clause, `=>`, and finally an expression. The type of a function expression is an arrow type consisting of the function's parameter type followed by the token `→`, followed by the function's return type, and the function's optional `throws` clause. Unlike declared functions (described in Chapter 12), function expressions are not allowed to include static parameters nor `where` clauses (described in Chapter 11).

Here is a simple example:

```
fn (x:Double) => if x < 0 then -x else x end
```

13.8 Operator Applications

Syntax:

```
OpExpr ::= EncloserOp OpExpr? EncloserOp?
         | OpExpr EncloserOp OpExpr?
         | Primary
Primary ::= LeftEncloser ExprList? RightEncloser
         | Primary [ExprList?]
         | Primary ^ BaseExpr
         | Primary ExponentOp
EncloserOp ::= Encloser
            | Op
```

To support a rich mathematical notation, Fortress allows most Unicode characters that are specified to be mathematical operators to be used as operators in Fortress expressions, as well as various tokens described in Chapter 16. Most of the operators can be used as prefix, infix, postfix, or nofix operators as described in Section 16.3; the fixity of an operator is determined syntactically, and the same operator may have definitions for multiple fixities.

Syntactically, an operator application consists of an operator and its argument expressions. If the operator is a prefix operator, it is followed by its argument expression. If the operator is an infix operator, its two argument expressions come both sides of the operator. If the operator is a postfix operator, it comes right after its argument expression. Like function calls, argument expressions are evaluated *in parallel* in separate implicit threads. After evaluation of arguments completes normally, the body of the operator definition is evaluated with the parameters of the operator bound to the values of the argument expressions. The value and the type of an operator application are the value and the type of the operator body.

Here are some examples:

```

(-b + sqrt(b2 - 4ac))/2a
nn e(-n) sqrt(2πn)
ak bn-k
x1 y2 - x2 y1
1/2gt2
n(n + 1)/2
(j + k)!/(j!k!)
1/3 3/5 5/7 7/9 9/11
17.3 meter/second
17.3 m/s
u · (v × w)
(A ∪ B) INTERSECT C
(A ∪ B) ∩ C
i < j ≤ k ∧ p < q
print("The answers are " (p + q) " and " (p - q))
⟨2, 3, 4, 5⟩

```

13.9 Object Expressions

Syntax:

```
DelimitedExpr ::= object Extends? GoInAnObject end
```

Object expressions denote object values. Syntactically, they start with `object`, followed by an optional `extends` clause, a series of field declarations, method declarations, or property declarations, and finally `end`. The type of an object expression is an anonymous object trait type that extends the traits listed in the `extends` clause of the object expression. The object trait type does not include the methods introduced by the object expression (i.e., those methods not provided by any supertraits of the object expression). Each object trait type is associated with a program location; every evaluation of a given object expression has the same anonymous object trait type. Two object expressions at different program locations with the same `extends` clause have different object trait types.

Unlike object declarations (described in Chapter 10), object expressions are not allowed to include modifiers, value parameters, static parameters, or `where` clauses (described in Chapter 11). Unlike object declarations, which must not include any free static variables (i.e., all static variables in an object declaration must occur either as a static parameter or as a `where`-clause variable), object expressions may include free static variables. Field initializers and methods may refer to any variables that are in scope in the context in which the object expression occurs. As with object declarations, initializers are evaluated in textual program order and may refer to previous fields; the object being constructed may not be referred to in any way.

For example, the following object expression:

```
f[[T]](x:T) = object f:T = x end
```

has a static variable *T* that is not its static parameter nor its `where`-clause variable.

The following example expression evaluates to a new object extending trait `List`:

```
object extends { List }
  first() = throw Error
  rest() = throw Error
  cons(x) = Cons(x, self)
  append(xs) = xs
end
```

13.10 Assignments

Syntax:

```
Expr          ::= AssignLefts AssignOp Expr
AssignLefts   ::= ( AssignLeft(, AssignLeft)* )
               | AssignLeft
AssignLeft    ::= SubscriptExpr
               | FieldSelection
               | BindId
SubscriptExpr ::= Primary [ExprList?]
FieldSelection ::= Primary . Id
BindId        ::= Id
AssignOp      ::= := | Op =
```

An assignment expression consists of a left-hand side (*AssignLefts*) indicating one or more variables, subscripted expressions (as described in Section 34.7), or field accesses to be updated, an assignment token, and a right-hand-side expression. Multiple left-hand sides must be grouped using tuple notation (comma-separated and parenthesized). Variables updated in an assignment expression must already be declared.

The assignment token ‘`:=`’ indicates ordinary assignment. Ordinary assignment proceeds in two phases. In the first, the evaluation phase, the right-hand-side expression is evaluated in parallel with each of the left-hand-side subexpressions, forming an implicit thread group. Evaluating a left-hand variable does nothing. Evaluating a left-hand field reference evaluates the receiving object. Evaluating a left-hand subscripting operation evaluates the receiving object and the index in parallel in a nested thread group. After the outer implicit thread group completes normally, the assignment phase begins. Each component of the right-hand-side value is assigned to the corresponding component of the left-hand side in parallel, forming an implicit thread group. Assigning a left-hand variable simply changes the binding for the variable’s location to contain the new value. Assigning a left-hand field reference calls the corresponding setter method on the receiver object, passing the new value. Assigning a left-hand subscript simply calls the subscripted assignment operation on the corresponding object with the new value.

Any operator (other than ‘`:`’ or ‘`=`’ or ‘`<`’ or ‘`>`’) followed by ‘`=`’ with no intervening whitespace indicates compound (updating) assignment. This adds an additional phase to assignment between the two phases of ordinary assignment. In the first phase, a left-hand field reference invokes the getter for the corresponding field after the receiving object has been evaluated. A left-hand subscripting operation invokes the subscripting operator on the receiving object once receiver and index have been evaluated. A left-hand variable simply returns the current value of the location associated with that variable. In the new second phase, the operator indicated in the assignment is invoked. The left-hand argument is the value of the left-hand expression evaluated in the first phase; the right-hand argument

is the value evaluated for the right-hand side of the assignment expression. When the operator evaluation completes normally, the assignment phase is run as in ordinary assignment.

The important point to understand is that compound assignment evaluates its subexpressions exactly once, and that the parts of assignment proceed implicitly in parallel where possible.

Consider the following assignment:

$$(a[i], b.x, c) \text{UPD} = f(t, u, v)$$

Here in the first phase we evaluate a and i in parallel, then when this completes we invoke the indexing method of a to evaluate $a[i]$. In parallel we evaluate b and then call the getter method $b.x$. In parallel we look up the value of variable c . Finally, we evaluate $f(t, u, v)$ in parallel with all of these left-hand sides. In the second phase, we combine the results using the UPD operator, to evaluate $(a_i, b.x, c) \text{UPD} f(t, u, v)$. This must return a tuple of three values (p, q, r) . In the final phase, in parallel we call the indexed assignment operator $a[i] := p$, call the setter $b.x := q$, and perform the local assignment $c := r$.

Here are some simpler and more commonplace examples of assignment:

```

x := f(0)
cij := cij + aikbkj
(a, b, c) := (b, c, a)           (* Permute a, b, and c *)
x += 1
(x, y) += (δx, δy)
myBag = myBag ∪ newItems
myBag ∪= newItems

```

13.10.1 Definite Assignment

References to uninitialized variables are statically forbidden. As with the Java Programming Language, this static constraint is ensured with a specific conservative flow analysis. In essence, an initialization of a variable must occur on every possible execution path to each reference to a variable.

13.11 Do Expressions

Syntax:

```

Do           ::= (DoFront also)* DoFront end
DoFront     ::= (at Expr)? atomic? do BlockElems?
BlockElems  ::= BlockElem+
BlockElem   ::= LocalVarFnDecl
              | Expr(, GeneratorList)?
LocalVarFnDecl ::= LocalFnDecl+
              | LocalVarDecl

```

A *do* expression consists of a series of *block expressions* separated by *also* and terminated by *end*. Each block expression is preceded by an optional *at* expression (described in Section 32.7), an optional *atomic*, and *do*. When prefixed by *at* or *atomic*, it is as though that block expression were evaluated as the body expression of an *at* or *atomic* expression respectively. A block expression consists of a series of *elements*: expressions, generated expressions (described in Section 13.11.2), local variable declarations, or local function declarations.

A single block expression evaluates its elements in order: each element must complete before evaluation of the next can begin, and the block expression as a whole does not complete until the final element completes. If the last element

of the block expression is an expression, the value and type of this expression are the value and type of the block expression as a whole. Otherwise, the value and type of the block expression is (). Each block expression introduces a new scope. Some compound expressions have clauses that are implicitly block expressions.

Here are examples of function declarations whose bodies are `do` expressions:

```
f(x : ℝ64) = do
  (sin(x) + 1)2
end
```

```
foo(x : ℝ64) = do
  y = x
  z = 2x
  y + z
end
```

```
mySum(i : ℤ64) : ℤ64 = do
  acc : ℤ64 := 0
  for j ← 0 : i do
    acc := acc + j
  end
  acc
end
```

13.11.1 Distinguishing a Local Declaration from an Equality Expression

Because a local declaration shares a syntax with an *equality expression*, we provide rules for disambiguation:

- If an expression of the form “ $e = e$ ” occurs as a proper subexpression in any non-block expression, it is an equality expression.
- If such an expression occurs as an immediate subexpression of a block expression, it is a local declaration. Adding parentheses makes the expression an equality expression.

13.11.2 Generated Expressions

If a subexpression of a `do` expression has type (), the expression may be followed by a ‘,’ and a generator list (described in Section 13.14). Writing “*expr, gens*” is equivalent to writing “`for gens do expr end`”. See Section 13.15 for the semantics of the `for` expression. Note in particular that *expr* can be a reducing assignment of the form “`var OP= expr`”.

13.11.3 Parallel Do Expressions

A series of blocks may be run in parallel using the `also` construct. Any number of contiguous blocks may be joined together by `also`. Each block is run in a separate implicit thread; these threads together form a group. The expression as a whole completes when the group is complete. A thread can be placed in a particular region by using an `at` expression as described in Section 32.7. When multiple block expressions are separated by `also`, each block expression must have type (); the result and type of the parallel `do` expression is also ().

For example:

```

treeSum(t : TreeLeaf) = 0
treeSum(t : TreeNode) = do
  var accum := 0
  do
    accum += treeSum(t.left)
  also do
    accum += treeSum(t.right)
  also do
    accum += t.datum
  end
  accum
end

```

Note the use of the reduction variable `accum` (Section 4.4.1) within the threads in the group.

13.12 Label and Exit

Syntax:

```

DelimitedExpr ::= label Id BlockElems end Id
FlowExpr      ::= exit Id? (with Expr)?

```

A block expression may be labeled using a `label` expression, which consists of `label`, an identifier (the block's *name*), a block expression (its *body*), `end`, and finally its name again (it is a static error if the identifier after `end` is not the name). The name of a `label` expression is in scope in the label namespace at any point in its body that is not within a `spawn` subexpression of the body. A `label` expression is evaluated by evaluating its body.

An `exit` expression consists of `exit`, an optional identifier (the *target*) and an optional `with` clause, which consists of `with` followed by an expression. If a target is specified, it is a static error if the target is not in scope in the label namespace at the `exit` expression. That is, the target must be the name of a statically enclosing `label` expression, and the `exit` expression must not be within a `spawn` expression that is contained in the `label` expression. If no target is specified, the target is implicitly the name of the smallest statically enclosing `label` expression; it is a static error if there is no such expression.

An `exit` expression with a `with` clause evaluates its `with` clause expression to yield an *exit value*. The `exit` expression completes abruptly with the exit value (see Section 4.2). An `exit` expression with no `with` clause has an implicit `with` clause whose expression is `()`. The type of an `exit` expression is `BottomType`.

If the evaluation of the body of a `label` expression completes normally, its value is the value of the body. If the evaluation of the body completes abruptly with an `exit` expression whose target is the name of the `label` expression, then the evaluation of the `label` expression completes normally and its value is the exit value of the `exit` expression. The type of a `label` expression is the union of the type of the last expression of its block expression and the types of the values of any `exit` expressions within the `label` expression whose target is the `label` expression's name.

If one or more `try` expressions are nested between an `exit` expression and the targeted `label` block, the `finally` clauses of these expressions are run in order, from innermost to outermost, as described in Section 13.25. If any `finally` clause completes abruptly by throwing an exception, the `exit` expression fails to exit, the evaluation of the `label` expression completes abruptly, and the exception is propagated.

Here is a simple example:

```

label I95
  if goingTo(Sun)
  then exit I95 with x32B

```

```

    else x32A
    end
end I95

```

The expression `exit I95 with x32B` completes abruptly and attempts to transfer control to the end of the targeted labeled block `label I95`. The targeted labeled block completes normally with value *x32B*.

13.13 While Loops

Syntax:

```
DelimitedExpr ::= while Expr Do
```

A `while` loop consists of a *condition expression* of type Boolean followed by a simple `do` expression (see Section 13.11). An iteration of a `while` loop evaluates the condition expression; if it completes normally and returns *true* it then evaluates the body block expression to completion. When one iteration completes a new one is run until either an iteration completes abruptly (in which case the evaluation of the `while` expression completes abruptly), or the condition expression has the value *false* (in which case the `while` loop completes normally with value `()`).

13.14 Generators

Syntax:

```

GeneratorList ::= Generator(, Generator)*
Generator     ::= Id ← Expr
                | (Id, IdList) ← Expr
                | Expr
IdList       ::= Id(, Id)*

```

Fortress makes extensive use of comma-separated *generator lists* to express parallel iteration. Generator lists occur in generated expressions (described in Section 13.11.2), `for` loops (described in Section 13.15), sums and big operators (described in Section 13.17), and comprehensions (described in Section 13.28). We refer to these collectively as *expressions with generators*. Every expression with generators contains a *body expression* which is evaluated for each combination of values bound in the generator list (each such combination yields an *iteration* of the body).

An element of a generator list is either a *generator binding* or a boolean expression. A generator binding consists of one or more comma-separated identifiers followed by the token `←`, followed by a subexpression (called the *generator expression*). A generator expression evaluates to an object whose type is `Generator`. A generator encapsulates zero or more *generator iterations*. By default, the programmer must assume that generator iterations are run in parallel in separate implicit threads unless the special *sequential* generator is used; the actual behavior of generators is dictated by library code, as described in Section 32.8. No generator iterations are run until the generator expression completes. For each generator iteration, a generator object produces a value or a tuple of values. These values are bound to the identifiers to the left of the arrow, which are in scope of subsequent generator list elements and of the body of the construct containing the generator list.

A boolean expression in a generator list is interpreted as a *filter*. A generator iteration is performed only if the result of the filter expression is *true*. If the filter is *false*, subsequent expressions in the generator list will not be evaluated.

The order of nesting of generators need not imply anything about the relative order of nesting of iterations. In most cases, multiple generators can be considered equivalent to multiple nested loops. However, the compiler will make an effort to choose the best possible iteration order it can for a multiple-generator loop, and may even combine generators together; there may be no such guarantee for nested loops. Thus loops with multiple generators are preferable in

general. Note that the early termination behavior of nested looping is subtly different from a single multi-generator loop, since nested loops give rise to nested thread groups; see Section 32.6.

Each generator iteration of the innermost generator list element corresponds to a *body iteration*, or simply an *iteration* of the generator list. Each iteration is run in its own implicit thread. Each expression in the generator list can each be considered to evaluate in a separate implicit thread. Together these implicit threads form a thread group. Evaluation of an expression with generators completes only when this thread group has completed.

Some common Generators include:

$l : u$	Any range expression
$a.indices$	The index set of an array a
$\{0, 1, 2, 3\}$	The elements of an aggregate expression
$sequential(g)$	A sequential version of generator g

The generator $sequential(g)$ forces the iterations using distinct values from g to be performed in order. Every generator has an associated *natural order* which is the order obtained by $sequential$. For example, a sequential `for` loop starting at 1 and going to n can be written as follows:

```
for i ← sequential(1:n) do
  ...
end
```

The *sequential* generator respects generator list ordering; it will always nest strictly inside preceding generator list elements and outside succeeding ones.

Given a multidimensional array, the *indices* generator returns a tuple of values, which can be bound by a tuple of variables to the left of the arrow:

```
(i, j) ← my2DArray.indices
```

The parallelism of a loop on this generator follows the spatial distribution (discussed in Section 32.5) of *my2DArray* as closely as possible.

13.15 For Loops

Syntax:

```
DelimitedExpr ::= for GeneratorList DoFront end
DoFront       ::= (at Expr)? (atomic)? do BlockElems?
```

A `for` loop consists of `for` followed by a generator list (discussed in Section 13.14), followed by a non-parallel `do` expression (the loop *body*; see Section 13.11). Parallelism in `for` loops is specified by the generators used (see Section 13.14); in general the programmer must assume that each loop iteration will occur independently in parallel unless every generator is explicitly *sequential*. For each iteration, the body expression is evaluated in the scope of the values bound by the generators. The body of a `for` expression can make use of reduction variables as described in Section 4.4.1. The value and type of a `for` loop is `()`.

13.16 Ranges

Syntax:

```
Range ::= Expr? : Expr?( : Expr?)?
       | Expr # Expr
```

A *range expression* is used to create a special kind of Generator for a set of integers, called a Range, useful for indexing an array or controlling a for loop. Generators in general are discussed further in Section 13.14.

An *explicit range* is self-contained and completely describes a set of integers. Assume that a , b , and c are expressions that produce integer values.

- The range $a : b$ is the set of $n = \max(0, b - a + 1)$ integers $\{a, a + 1, a + 2, \dots, b - 2, b - 1, b\}$. This is a *nonstrided* range. If a and b are both static expressions (described in Section 13.26), then it is a *static range* of type `StaticRange[[a, n, 1]]` and therefore also a *range of static size* of type `RangeOfStaticSize[[n]]`.
- The range $a : b : c$ is the set of $n = \max(0, \lfloor \frac{b-a+c}{c} \rfloor)$ integers $\{a, a + c, a + 2c, \dots, a + \lfloor \frac{b-a}{c} \rfloor c\}$, unless c is zero, in which case it throws an exception. (If c is a static expression, then it is a static error if c is zero.) This is a *strided* range. If a , b , and c are all static expressions, then it is a *static range* of type `StaticRange[[a, n, c]]` and therefore also a *range of static size* of type `RangeOfStaticSize[[n]]`.
- The range $a \# n$ is the set of $\max(0, n)$ integers $\{a, a + 1, a + 2, \dots, a + n - 3, a + n - 2, a + n - 1\}$. This is a *nonstrided* range. If a and n are both static expressions, then it is a *static range* of type `StaticRange[[a, n, 1]]`. If n is a static expression, then it is a *range of static size* of type `RangeOfStaticSize[[n]]`, even if a is not a static expression.

Non-static components of a range expression are computed in separate implicit threads. The range is constructed when all components have completed normally.

An *implicit range* may be used only in certain contexts, such as array subscripts, that can supply implicit information. Suppose an implicit range is used as a subscript for an axis of an array for which the lower bound is l and the upper bound is u .

- The implicit range $:$ is treated as $l : u$.
- The implicit range $: : c$ is treated as $l : u : c$.
- The implicit range $: b$ is treated as $l : b$.
- The implicit range $: b : c$ is treated as $l : b : c$.
- The implicit range $a :$ is treated as $a : u$.
- The implicit range $a : : c$ is treated as $a : u : c$.

One may test whether an integer is in a range by using the operator \in :

```
if j ∈ a : b then print "win" end
```

Ranges may be compared as if they were sets of integers by using \subset (SUBSET) and \subseteq (SUBSETEQ) and $=$ and \supseteq (SUPSETEQ) and \supset (SUPSET).

Ranges may be intersected using the operator \cap (INTERSECTION).

The size of a range (the number of integers in the set) may be found by using the set-cardinality operator $|\dots|$. For example, the value of $|3 : 7|$ is 5 and the value of $|1 : 100 : 2|$ is 50.

Note that a range is very different from an interval with integer endpoints. The range $3 : 5$ contains only the values 3, 4, and 5, whereas the interval $[3, 5]$ contains all real numbers x such that $3 \leq x \leq 5$.

13.17 Summations and Other Reduction Expressions

Syntax:

$FlowExpr ::= Accumulator ([GeneratorList])? Expr$
 $Accumulator ::= \sum | \prod | BIG Op$

A *reduction expression* begins with a big operator such as \sum or \prod followed by an optional generator list (described in Section 13.14), followed by a body expression. A complete list of these operators are described in Section 16.8.1; each reduction operator corresponds to a binary operator OP which extends $Monoid[[T, OP]]$, which is $+$ in the case of \sum , juxtaposition in case of \prod , and Op in the case of $BIG Op$. When a generator list is provided, generators produce values and bind the values to the identifiers that are used in the subexpression. Each iteration of the body expression is assumed to be evaluated in a separate implicit thread as described in Section 13.14. The resulting values are combined together using OP as with reduction variables (see Section 4.4.1). The value of a reduction expression is this combined value. The type of a reduction expression is the return type of the big operator used (which will match the argument and return types of OP).

When no generator list is provided, the body is taken to be an object of type `Generator` and the elements it generates are combined. Thus, the reduction expression $\sum a$ is equivalent to $\sum_{x \leftarrow a} x$.

A reduction expression with a generator list:

$$\sum[v_1 \leftarrow g_1, v_2 \leftarrow g_2, \dots]e$$

is equivalent to the following code:

```

do
  result = 0
  for v1 ← g1, v2 ← g2, ... do
    result += e
  end
  result
end

```

where *result* is a fresh variable. A reduction expression without a generator list:

$$\sum g$$

is equivalent to the following:

$$\sum[x \leftarrow g]x$$

Note that reduction expressions without generator lists can be used to conveniently sum any aggregate expression (described in Section 13.27), since every aggregate expression is a generator.

13.18 If Expressions

Syntax:

$DelimitedExpr ::= \text{if } Expr \text{ then } BlockElems \text{ Elifs? Else? end}$
 $\quad \quad \quad | \quad (\text{if } Expr \text{ then } BlockElems \text{ Elifs? Else end?})$
 $Elifs ::= Elif^+$
 $Elif ::= \text{elif } Expr \text{ then } BlockElems$
 $Else ::= \text{else } BlockElems$

An *if expression* consists of `if` followed by a condition expression of type `Boolean`, followed by `then`, a block expression, an optional sequence of `elif` clauses (each consisting of `elif` followed by a condition expression, `then`, and a block expression), an optional `else` clause (consisting of `else` followed by a block expression), and finally `end`. Each clause forms a block expression and has the various properties of block expressions (described

in Section 13.11). An `if` expression first evaluates its condition expression. If the condition expression completes normally and results in *true*, the `then` clause is evaluated. If the condition expression results in *false*, the next clause (either `elif` or `else`), if any, is evaluated. An `elif` clause works just as the original `if`, evaluating its condition expression and continuing with its `then` clause if the condition is *true*. An `else` clause simply evaluates its block expression. The type of an `if` expression is the union of the types of the block expressions of each clause. If there is no `else` clause in an `if` expression, then every clause must have type `()`. The result of the `if` expression is the result of the block expression which is evaluated; if no block expression is evaluated it is `()`. The reserved word `end` may be elided if the `if` expression is immediately enclosed by parentheses. In such a case, an `else` clause is required.

For example,

```
if x ∈ {0, 1, 2} then 0
elif x ∈ {3, 4, 5} then 3
else 6 end
```

13.19 Case Expressions

Syntax:

```
DelimitedExpr ::= case Expr Op? of CaseClauses CaseElse? end
CaseClauses  ::= CaseClause+
CaseClause   ::= Expr ⇒ BlockElems
CaseElse     ::= else ⇒ BlockElems
```

A case expression begins with `case` followed by a condition expression, followed by an optional operator, `of`, a sequence of case clauses (each consisting of a *guarding expression* followed by the token `⇒`, followed by a block expression), an optional `else` clause (consisting of `else` followed by the token `⇒`, followed by a block expression), and finally `end`.

A case expression evaluates its condition expression and checks each case clause to determine which case clause matches. To find a matched case clause, the guarding expression of each case clause is evaluated in order and compared to the value of the condition expression. For the first clause, the condition expression and guarding expression are evaluated in separate implicit threads; for subsequent clauses the value of the condition expression is retained and only the guarding expression is evaluated. Once both guard and condition expressions have completed normally, the two values are compared according to an optional operator specified. If the operator is omitted, it defaults to `=` or `∈`. If the condition expression has type `Generator` or if the guarding expression does not, then the default operator is `=`; otherwise, it is `∈`. It is a static error if the specified operator is not defined for these types or if the operator's return type is not `Boolean`.

If the operator completes normally and returns *true*, the corresponding block expression is evaluated (see Section 13.11) and its value is returned. If the operator returns *false* matching continues with the next clause. If no matched clause is found, a `MatchFailure` exception is thrown. The optional `else` clause always matches without requiring a comparison. The value of a case expression is the value of the right-hand side of the matched clause. The type of a case expression is the union type of the types of all right-hand sides of the case clauses.

For example, the following case expression specifies the operator `∈`:

```
case planet ∈ of
  { Mercury, Venus, Earth, Mars } ⇒ "inner"
  { Jupiter, Saturn, Uranus, Neptune } ⇒ "outer"
  else ⇒ "remote"
end
```

but the following does not:

```

case 2 + 2 of
  4 ⇒ “it really is 4”
  5:7 ⇒ “we were wrong again”
end

```

13.20 Extremum Expressions

Syntax:

```

DelimitedExpr ::= case (largest | smallest) Op? of CaseClauses end
CaseClauses   ::= CaseClause+
CaseClause    ::= Expr ⇒ BlockElems

```

An extremum expression uses the same syntax as a `case` expression (described in Section 13.19) except that `largest` or `smallest` is used where a `case` expression would have a condition expression and an extremum expression does not have an optional `else` clause.

All guarding expressions of an extremum expression are evaluated in separate implicit threads. To find the largest (or smallest) quantity, the values of the guarding expressions are compared in parallel according to an optional operator specified. If the operator is omitted, it defaults to `CMP`. The union of the types of all the candidate expressions must be a subtype of `TotalOrderOperators`[[T , $<$, \leq , \geq , $>$, `CMP`]] for some T , $<$, \leq , \geq , and $>$, if a default operator is used (see Section 37.2 for details about the `TotalOrderOperators` trait). If an explicit operator is used, the explicit operator replaces `CMP` in `TotalOrderOperators`[[T , $<$, \leq , \geq , $>$, `CMP`]]. The invocations of `CMP` occur in parallel in separate implicit threads as part of the same group in which the guarding expressions themselves are evaluated, in a manner analogous to reduction (see Section 4.4.1). Which pairs of guarding expressions are compared is unspecified, except that the pairwise comparisons performed will be sufficient to determine that the chosen clause is indeed largest or smallest assuming a total order. Any or all pairwise comparisons may be considered.

The block expression of the clause with the largest (smallest) guarding expression (and only that clause) is evaluated. If more than one guarding expressions are tied for largest (smallest), the first clause in textual order is evaluated to yield the result of the extremum expression. The type of an extremum expression is the union of the types of all right-hand sides of the clauses.

For example, the following code:

```

case largest of
  1 mile ⇒ “miles are larger”
  1 kilometer ⇒ “we were wrong again”
end

```

evaluates to “miles are larger”. A more interesting example is described in Section 6.5.

13.21 Typecase Expressions

Syntax:


```

DelimitedExpr ::= typecase TypecaseBindings of TypecaseClauses CaseElse? end
TypecaseBindings ::= ( BindingList )
                    | Binding
                    | Id
BindingList ::= Binding(, Binding)*
Binding ::= BindId = Expr
TypecaseClauses ::= TypecaseClause+
TypecaseClause ::= TypecaseTypeRefs ⇒ BlockElems
TypecaseTypeRefs ::= ( TypeRefList )
                    | TypeRef
CaseElse ::= else ⇒ BlockElems

```

A `typecase` expression begins with `typecase` followed by a sequence of bindings (either an identifier or a sequence of an identifier followed by the token `=`, followed by an expression), followed by `of`, a sequence of typecase clauses (each consisting of a sequence of *guarding types* followed by the token `⇒`, followed by a block expression), an optional `else` clause (consisting of `else` followed by the token `⇒`, followed by a block expression), and finally `end`.

A `typecase` expression evaluates its bindings in separate implicit threads; if they complete normally, each subexpression is bound to the corresponding identifier and the first matching typecase clause is chosen. A typecase clause matches if the type of every identifier bound in the bindings is a subtype of the corresponding type in the clause. The block expression of the first matched clause (and only that clause) is evaluated (see Section 13.11) to yield the value of the `typecase` expression. If no matched clause is found, a `MatchFailure` exception is thrown. The type of a `typecase` expression is the union type of the types of all right-hand sides of the typecase clauses. When typechecking a clause, an identifier bound in the binding is assumed to have the static type specified by the corresponding type in the clause. This type will in general be more specific than the static type of the expression to which the variable was bound.

For example:

```

typecase x = myLoser.myField of
  String ⇒ x.append("f○○")
  Number ⇒ x + 3
  Object ⇒ yogiBerraAutograph
end

```

Note that “*x*” has a different type in each clause.

13.22 Atomic Expressions

Syntax:

```

FlowExpr ::= atomic AtomicBack
            | tryatomic AtomicBack
AtomicBack ::= AssignLefts AssignOp Expr
              | OpExpr
              | DelimitedExpr

```

As Fortress is a parallel language, an executing Fortress program consists of a set of threads (See Section 4.4 for a discussion of parallelism in Fortress). In multithreaded programs, it is often convenient for a thread to evaluate some expressions *atomically*. For this purpose, Fortress provides atomic expressions.

An atomic expression consists of `atomic` followed by a *body expression*. Evaluating an atomic expression is simply evaluating the body expression. All reads and all writes which occur as part of this evaluation will appear to

occur simultaneously in a single atomic step with respect to *any* action performed by any thread which is dynamically outside. This is specified in detail in Chapter 21. The value and type of an `atomic` expression are the value and type of its body expression.

A `tryatomic` expression consists of `tryatomic` followed by an expression. See Section 32.3 for a discussion of `tryatomic` expressions.

A function or method with the modifier `atomic` acts as if its entire body were surrounded in an `atomic` expression. However, it is a static error if an API declares a functional f with the modifier `atomic` but a component implementing the API defines f whose body is an `atomic` expression without the modifier. Functionals with the modifier `io` cannot be called within an `atomic` expression. Thus, a functional must not have both `atomic` and `io` modifiers.

When the body of an `atomic` expression completes abruptly, the `atomic` expression completes abruptly in the same way. If it completes abruptly by exiting to an enclosing `label` expression, writes within the block are retained and become visible to other threads. If it completes abruptly by throwing an uncaught exception, all writes to objects allocated before the `atomic` expression began evaluation are discarded. Writes to newly allocated objects are retained. Any variable reverts to the value it held before evaluation of the `atomic` expression began. Thus, the only values retained from the abruptly completed `atomic` expression will be reachable from the exception object through a chain of newly allocated objects.

Atomic expressions may be nested arbitrarily; the above semantics imply that an inner `atomic` expression is atomic with respect to evaluations which occur dynamically outside the inner `atomic` expression but dynamically inside an enclosing `atomic`.

Implicit threads may be created dynamically within an `atomic` expression. These implicit threads will complete before the `atomic` expression itself does so. The implicit threads may run in parallel, and will see one another's writes; they may synchronize with one another using nested `atomic` expressions.

Note that `atomic` expressions may be evaluated in parallel with other expressions. An `atomic` expression experiences *conflict* when another thread attempts to read or write a memory location which is accessed by the `atomic` expression. The evaluation of such an expression must be partially serialized with the conflicting memory operation (which might be another `atomic` expression). The exact mechanism by which this occurs will vary; the necessary serialization is provided by the implementation. In general, the evaluation of a conflicting `atomic` expression may be abandoned, forcing the effects of execution to be discarded and execution to be retried. The longer an `atomic` expression evaluates and the more memory it touches the greater the chance of conflict and the larger the bottleneck a conflict may impose.

For example, the following code uses a shared counter atomically:

```
sum : N := 0
accumArray[[N extends Additive, nat x]](a : N[x]) : () =
  for i ← a.indices do
    atomic sum += a[i]
  end
```

The loop body reads $a[i]$ and sum , then adds them and writes the result back to sum ; this will appear to occur atomically with respect to all other threads—including both other iterations of the loop body and other simultaneous calls to `accumArray`. Note in particular that the `atomic` expression will appear atomic with respect to reads and writes not in `atomic` expressions of $a[i]$ and sum .

13.23 Spawn Expressions

Syntax:

```
FlowExpr ::= spawn Expr
```

A spawned thread is created using a `spawn` expression. A `spawn` expression consists of `spawn` followed by an expression. A `spawn` expression spawns a thread which evaluates its subexpression in parallel with any succeeding evaluation. The value of a `spawn` expression is the spawned thread and the type of the expression is `Thread[[T]]`, where `T` is the static type of the expression spawned. A `spawn` expression constitutes an `io` action, and thus cannot be run within the body of an `atomic` expression. The semantics of spawned threads are discussed in Section 4.4.

13.24 Throw Expressions

Syntax:

```
FlowExpr ::= throw Expr
```

A `throw` expression consists of `throw` followed by a subexpression. The subexpression must have the type `Exception` (see Chapter 14). A `throw` expression evaluates its subexpression to an exception value and throws the exception value; the expression completes abruptly and has `BottomType`.

The type `Exception` has exactly two direct mutually exclusive subtypes, `CheckedException` and `UncheckedException`. Every `CheckedException` that is thrown must be caught or forbidden by an enclosing `try` expression (see Section 13.25), or it must be declared in the `throws` clause of an enclosing functional declaration (see Section 12.1). Similarly, every `CheckedException` declared to be thrown in the static type of a functional called must be either caught or forbidden by an enclosing `try` expression, or declared in the `throws` clause of an enclosing functional declaration.

13.25 Try Expressions

Syntax:

```
DelimitedExpr ::= try BlockElems Catch? (forbid TraitTypes?)? (finally BlockElems)? end  
Catch ::= catch Id CatchClauses  
CatchClauses ::= CatchClause+  
CatchClause ::= TraitType ⇒ BlockElems
```

A `try` expression starts with `try` followed by a block expression (the `try block`), followed by an optional `catch` clause, an optional `forbid` clause, an optional `finally` clause, and finally `end`. A `catch` clause consists of `catch` followed by an identifier, followed by a sequence of subclauses (each consisting of an exception type followed by the token `⇒` followed by a block expression). A `forbid` clause consists of `forbid` followed by a set of exception types. A `finally` clause consists of `finally` followed by a block expression. Note that the `try block` and the clauses form block expressions and have the various properties of block expressions (described in Section 13.11).

The expressions in the `try block` are first evaluated in order until they have all completed normally, or until one of them completes abruptly. If the `try block` completes normally, the *provisional* value of the `try` expression is the value of the last expression in the `try block`. In this case, and in case of exiting to an enclosing `label` expression, the `catch` and `forbid` clauses are ignored.

If an expression in the `try block` completes abruptly by throwing an exception, the exception value is bound to the identifier specified in the `catch` clause, and the type of the exception is matched against the subclauses of the `catch` clause in turn, exactly as in a `typecase` expression (Section 13.21). The right-hand-side block expression of the first matching subclause is evaluated. If it completes normally, its value is the provisional value of the `try` expression. If the `catch` clause completes abruptly, the `try` expression completes abruptly. If a thrown exception is not matched by the `catch` clause (or this clause is omitted), but it is a subtype of the exception type listed in a `forbid` clause, a new `ForbiddenException` is created with the thrown exception as its argument and thrown. The exception thrown by the `try block` is *chained* to the `ForbiddenException` as described in Section 14.3.

If an exception thrown from a `try` block is matched by both `catch` and `forbid` clauses, the exception is caught by the `catch` clause. If an exception thrown from a `try` block is not matched by any `catch` or `forbid` clause, the `try` expression completes abruptly.

The `finally` clause is evaluated after completion of the `try` block and any `catch` or `forbid` clause. The expressions in the `finally` clause are evaluated in order until they have all completed normally, or until one of them completes abruptly. In the latter case, the `try` expression completes abruptly exactly as the subexpression in the `finally` clause does.

If the `finally` clause completes normally, and the `try` block or the `catch` clause completes normally, then the `try` expression completes normally with the provisional value of the `try` expression. Otherwise, the `try` expression completes abruptly as specified above.

For example, the following `try` expression:

```
try
  inp = read(file)
  write(inp, newFile)
forbid IOException
end
```

is equivalent to:

```
try
  inp = read(file)
  write(inp, newFile)
catch e
  IOException => throw ForbiddenException(e)
end
```

The following example ensures that `file` is closed properly even if an IO error occurs:

```
try
  open(file)
  inp = read(file)
  write(inp, newFile)
catch e
  IOException => throw ForbiddenException(e)
finally
  close(file)
end
```

13.26 Static Expressions

Static expressions denote *static values*. Conceptually, a static expression is not evaluated while a program is running, and thus its evaluation cannot complete abruptly. Given instantiations of all static parameters (described in Chapter 11) in scope of a static expression, the value of the static expression can be determined statically. Static expressions which use a restricted subset of operations can occur as *static arguments* which instantiate static parameters and `where`-clause constraints (described in Section 11.6). We define the set of static expressions by first defining the types of static expressions, and distinguishing static values from the closely related literal values. We then describe the expressions that evaluate to the various kinds of static values.

13.26.1 Types of Static Expressions

There are three groups of traits that describe literals and static expressions:

1. The *literal* traits, which describe boolean, character, string, and numerals. For example, the literal *true* has trait `BooleanLiteral[true]` and a character literal has trait `Character`. See Section 13.1 for a discussion of Fortress literals.
2. The *constant* traits, which describe values denoted by expressions composed from literals and operators (described in Section 13.26.2); the type of a constant expression encodes the value of the expression. For example, the type of a constant expression $3 + 5$, `IntegerConstant[false, 3 + 5]`, encodes the value of the expression; the operator `+` is used in the static arguments. When an operator is not supported in static arguments, the Fortress standard libraries can still encode the value of the expression by providing a specific constant trait such as `RationalValueTimesPi[false, 1, 1]` (described in Section 13.1.1).
3. The *static* traits, which describe values denoted by expressions composed from literals, operators (described in Section 13.26.2), and `bool`, `nat`, and `int` parameters; here the type does not encode the value of the expression, but the value of the expression can nevertheless be known statically if specific values are specified for the static parameters. Also, in situations where the type of an expression composed solely from literals and operators nevertheless cannot be described by a constant trait, then a static trait may be used to describe it instead. For example, a static expression $2(3 + m)$ where *m* is a `nat` parameter has trait `NaturalStatic`.

The only operation on literals that produces a new literal (as opposed to a constant) is concatenation by using the operator `||`. One may concatenate mixed string and character literals, producing a string literal. For example, `"foo" || "bar"` yields `"foobar"`. One may also concatenate two natural numerals of the same radix, producing a new natural numeral of that radix. For example, `deadc16 || 0de16` yields `deadc0de16`.

Every literal trait extends an appropriate constant trait, and every constant trait extends an appropriate static trait. So every literal is also a constant expression, and every constant expression is a static expression.

13.26.2 Static Expressions and Values

Static parameters are static expressions. A `nat` parameter denotes a value that has type `NaturalStatic` (which extends `IntegerStatic`). An `int` parameter denotes a value that has type `IntegerStatic`. A `bool` parameter denotes a value that has type `BooleanStatic`.

Boolean static expressions may be combined using the operators `∧`, `∨`, `⊕`, `≡`, `↔`, `NAND`, `NOR`, `=`, `≠`, and `→` (See Appendix F for a discussion of Fortress operators.) to produce other static expressions denoting boolean static expressions. If both operands are boolean constant expressions, then the result is also a boolean constant expression.

Character static expressions may be compared using the operators `<`, `≤`, `≥`, `>`, `=`, and `≠` to produce boolean static expressions. If both operands are character constant expressions, then the result is a boolean constant expression.

Numeric static expressions may be compared using the operators `<`, `≤`, `≥`, `>`, `=`, and `≠` to produce boolean static expressions. If both operands are numeric constant expressions, then the result is a boolean constant expression.

Numeric static expressions may be combined using the operators and functions `+`, `-`, `×`, `·`, `/`, `!`, `MIN`, `MAX`, `√`, `floor`, `ceiling`, `hyperfloor`, `hyperceiling`, `gcd`, `lcm`, `sin`, `cos`, `tan`, `arcsin`, `arccos`, and `arctan` to produce new numeric static expressions. If the result is indeed a numeric static expression, and both operands are numeric constant expressions, then the result is also a numeric constant expression as long as the constant traits provided by the Fortress standard libraries can encode the value of the expression.

13.27 Aggregate Expressions

Syntax:

```

Primary      ::= LeftEncloser ExprList? RightEncloser
ExprList    ::= Expr(, Expr)*
Literal     ::= { Entry(, Entry)* }
              | [ RectElements ]
Entry       ::= Expr ↦ Expr
RectElements ::= Expr MultiDimCons*
MultiDimCons ::= RectSeparator Expr
RectSeparator ::= ;+
              | Whitespace

TupleExpr   ::= ( (Expr,)* (Expr ... ,)? Binding(, Binding)* )
              | NoKeyTuple
NoKeyTuple  ::= ( (Expr,)* Expr ... )
              | ( (Expr,)* Expr )
Binding     ::= BindId = Expr
BindId     ::= Id
              | -

```

Aggregate expressions evaluate to values that are themselves homogeneous collections of values. Each subexpression of an aggregate expression is evaluated in parallel in a separate implicit thread (see Section 4.4). With the exception of array expressions, map expressions, and tuple expressions, the resulting values are passed to the appropriate bracketing operator, which is a function with a varargs parameter constructing the aggregate. The former three aggregate operations construct the aggregate directly, after all subexpressions have completed normally. Any aggregate expression may reserve storage for its result before its elements complete evaluation.

Functions defining aggregate expressions are provided in the Fortress standard libraries for sets, maps, lists, tuples, matrices, vectors, and arrays.

13.27.1 Set Expressions

Set element expressions are enclosed in braces and separated by commas. The type of a set expression is $\text{Set}[[T]]$, where T is the union type of the types of all element expressions of the set expression.

Set containment is checked with the operator \in and the subset relationship is checked with the operator \subseteq . For example:

$$3 \in \{0, 1, 2, 3, 4, 5\}$$

evaluates to *true* and

$$\{0, 1, 2\} \subseteq \{0, 3, 2\}$$

evaluates to *false*.

13.27.2 Map Expressions

Map entries are enclosed in curly braces, separated by commas, and matching pairs are separated by \mapsto . The type of a map expression is $\text{Map}[[S, T]]$ where S is the union type of the types of all left-hand-side expressions of the map

entries, and T is the union type of the types of all right-hand-side expressions of the map entries. This type can be abbreviated as $\{S \mapsto T\}$.

A map m is indexed by placing an element in the domain of m enclosed in brackets immediately after an expression evaluating to m . Thus, the index is rendered as a subscript. For example, if:

$$m = \{ 'a' \mapsto 0, 'b' \mapsto 1, 'c' \mapsto 2 \}$$

then $m, 'b'$ evaluates to 1. In contrast, $m, 'x'$ throws a `NotFound` exception, as $'x'$ is not an index of m .

13.27.3 List Expressions

List element expressions are enclosed in angle brackets \langle and \rangle and are separated by commas. The type of a list expression is $\text{List}[T]$ where T is the union type of the types of all element expressions. This type can be abbreviated as $\langle T \rangle$.

A list l is indexed by placing an index enclosed in square brackets immediately after an expression evaluating to l . Thus, the index is rendered as a subscript. Lists are always indexed from 0. For example:

$$\langle 3, 2, 1, 0 \rangle_2$$

evaluates to 1.

13.27.4 Array Expressions

Array element expressions are enclosed in brackets. Element expressions along a row are separated only by whitespace. Two dimensional array expressions are written by separating rows with newlines or semicolons. If a semicolon appears, whitespace before and after the semicolon is ignored. The parts of higher-dimensional array expressions are separated by repeated-semicolons, where the dimensionality of the result is equal to one plus the number of repeated semicolons. The type of a k -dimensional array expression is $\text{Array}[T][n_0, \dots, n_{k-1}]$, where T is the union type of the types of the element expressions and n_0, \dots, n_{k-1} are the sizes of the array in each dimension. This type can be abbreviated as $T[n_0, \dots, n_{k-1}]$.

A k -dimensional array A is indexed by placing a sequence of k indices enclosed in brackets, and separated by commas, after an expression evaluating to A . Thus, the index is rendered as a subscript. By default arrays are indexed from 0. The horizontal dimension of an array is the last dimension mentioned in the array index. For example:

$$A = [1\ 2\ 3; 4\ 5\ 6; 7\ 8\ 9]$$

then $A_{1,0}$ evaluates to 4.

An array of two dimensions whose elements are a subtype of `Number` can be coerced to a matrix. Matrix types are written $\text{Matrix}[T][n_0 \times \dots \times n_{k-1}]$, where $k > 1$. A type of this form can be abbreviated as $T^{n_0 \times \dots \times n_{k-1}}$. Matrices are indexed in the same manner as arrays.

A one-dimensional array whose elements are a subtype of `Number` can be coerced to a vector. Vector types are written $\text{Vector}[T][n]$. A type of this form can be abbreviated as T^n , unless T is declared in the enclosing scope to be a physical dimension or unit.

The element expressions in an array expression may be either scalars or array expressions themselves. If an element is an array expression, it is “flattened” (pasted) into the enclosing expression. This pasting works because arrays never contain other arrays as elements. The elements along a row (or column) must have the same number of columns (or rows), though two elements in different rows (columns) need not have the same number of columns (rows). See Section 6.5 for a discussion of matrix unpasting.

The following four examples are all equivalent:

[3 4
5 6]

[3 4 ;
5 6]

[3 4
; 5 6]

[3 4 ; 5 6]

Here is a $3 \times 3 \times 3 \times 2$ matrix example:

```
[1 0 0
 0 1 0
 0 0 1 ; ; 0 1 0
           1 0 1
           0 1 0 ; ; 1 0 1
                           0 1 0
                           1 0 1

; ; ;
1 0 0
0 1 0
0 0 1 ; ; 0 1 0
           1 0 1
           0 1 0 ; ; 1 0 1
                           0 1 0
                           1 0 1]
```

13.27.5 Tuple Expressions

Tuple element expressions are enclosed in parentheses and separated by commas. Each element is one of:

- A plain expression “ e ”
- A varargs expression “ $e \dots$ ”
- A keyword-value pair “ $identifier = e$ ”

The following restrictions apply: No two keyword-value pairs may have the same keyword. No keyword-value pair may precede a plain expression. No varargs expression may follow a keyword-value pair or precede a plain expression. There must be at least one item. If there is exactly one item, it must be a varargs expression or a keyword-value pair (because an expression “ (e) ” is simply a parenthesized expression, not a tuple.) Also, there can be at most one item with a varargs expression. In a varargs expression, the expression “ e ” must be of type `OrderedGenerator[[T]]` for some T .

The type of a tuple expression is a tuple type (as discussed in Section 8.4), which may be described by taking the tuple expression and replacing each element expression with its type, except that a varargs expression having type `OrderedGenerator[[T]]` is replaced by $T \dots$ (for the most specific T possible). The element expressions are all evaluated before the tuple is constructed, and if there is a varargs expression then the generator is used to construct a `HeapSequence` (described in Section 40.3) containing all the generated values; this `HeapSequence` then becomes an element of the tuple. Tuples are value objects. There are no explicit destructors for tuples except multiple variable declarations as discussed in Section 6.3.

13.27.6 Distinguishing a Keyword-Value Pair from an Equality Expression

Because a keyword-value pair shares a syntax with an *equality expression*, we provide rules for disambiguation:

- If an expression “ $identifier = e$ ” has no parentheses around it, then it is an equality expression unless it is part of a tuple expression with more than one element expression.

- If the expression is in immediately surrounding parentheses with no other expression in the parentheses, then it is an equality expression unless the parenthesized expression is part of a juxtaposition sequence and is to be used as an argument to a function, in which case the parenthesized expression is a tuple expression.
- Adding parentheses makes the expression an equality expression.
- In the rare situations where “ $(identifier = e)$ ” is treated as an equality expression and it must be a tuple expression, “ $tuple(identifier = e)$ ” makes it a tuple expression where “ $tuple$ ” is an identity function defined in the Fortress standard libraries (as described in Section 13.31.4).

13.28 Comprehensions

Syntax:

<i>Comprehension</i>	::=	{ <i>Expr</i> <i>GeneratorList</i> }
		{ <i>Entry</i> <i>GeneratorList</i> }
		⟨ <i>Expr</i> <i>GeneratorList</i> ⟩
		[<i>ArrayComprehensionClause</i> ⁺]
<i>Entry</i>	::=	<i>Expr</i> ↦ <i>Expr</i>
<i>ArrayComprehensionClause</i>	::=	<i>ArrayComprehensionLeft</i> <i>GeneratorList</i>
<i>ArrayComprehensionLeft</i>	::=	<i>IdOrInt</i> ↦ <i>Expr</i>
		(<i>IdOrInt</i> , <i>IdOrIntList</i>) ↦ <i>Expr</i>
<i>IdOrInt</i>	::=	<i>Id</i>
		<i>IntLiteral</i>
<i>IdOrIntList</i>	::=	<i>IdOrInt</i> (, <i>IdOrInt</i>)*

Fortress provides *comprehension* syntax for several aggregate expressions (described in Section 13.27). A generator list binds values used in the body expression on the left-hand side of the token |. As described in Section 13.14, each iteration of the body expression must be assumed to execute in its own implicit thread. Comprehensions evaluate to aggregate values and have corresponding aggregate types. The rules for evaluation of a comprehension are similar to those for a reduction expression (see Section 13.17); examples can be found in Section 32.8.

A set comprehension is enclosed in braces, with a left-hand body separated by the token | from a generator list. For example, the comprehension:

$$\{ x^2 \mid x \leftarrow \{0, 1, 2, 3, 4, 5\}, x \text{ MOD } 2 = 0 \}$$

evaluates to the set

$$\{0, 4, 16\}$$

Map comprehensions are like set comprehensions, except that the left-hand body must be of the form $e_1 \mapsto e_2$. An exception is thrown if e_1 produces the same value but e_2 a different value on more than one iteration of the generator list. For example:

$$\{ x^2 \mapsto x^3 \mid x \leftarrow \{0, 1, 2, 3, 4, 5\}, x \text{ MOD } 2 = 0 \}$$

evaluates to the map

$$\{0 \mapsto 0, 4 \mapsto 8, 16 \mapsto 64\}$$

List comprehensions are like set comprehensions, except that they are syntactically enclosed in angle brackets. For example:

$$\langle x^2 \mid x \leftarrow \{0, 1, 2, 3, 4, 5\}, x \text{ MOD } 2 = 0 \rangle$$

evaluates to the list

$\langle 0, 4, 16 \rangle$

Array comprehensions are like set comprehensions, except that they are syntactically enclosed in brackets, and the left-hand body must be of the form $(index_1, index_2, \dots, index_n) \mapsto e$. Moreover an array comprehension may have multiple clauses separated by semicolons or line breaks. Each clause conceptually corresponds to an independent loop. Clauses are run in order. The result is an n -dimensional array. For example:

$$a = [(x, y, 1) \mapsto 0.0 \mid x \leftarrow 1 : xSize, y \leftarrow 1 : ySize \\ (1, y, z) \mapsto 0.0 \mid y \leftarrow 1 : ySize, z \leftarrow 2 : zSize \\ (x, 1, z) \mapsto 0.0 \mid x \leftarrow 2 : xSize, z \leftarrow 2 : zSize \\ (x, y, z) \mapsto x + y \cdot z \mid x \leftarrow 2 : xSize, y \leftarrow 2 : ySize, z \leftarrow 2 : zSize]$$

13.29 Type Ascription

Syntax:

$$Expr ::= Expr \text{ as } TypeRef$$

An expression consisting of a single subexpression, followed by `as`, followed by a type, is a *type ascription*. The value of the expression is the value of the subexpression. The static type of the expression is the ascripted type. The type of the subexpression must be a subtype of the ascripted type. A type ascription does not affect the dynamic type of the value the expression evaluates to (unlike a type assumption described in Section 13.30). Type ascription can be used to provide type information when type inference (see Chapter 20) cannot infer a type for an expression.

13.30 Type Assumption

Syntax:

$$Expr ::= Expr \text{ asif } TypeRef$$

An expression consisting of a single subexpression, followed by `asif`, followed by a type, is a *type assumption*. The value of the expression is the value of the subexpression. The static type of the expression is the given type. The type of the subexpression must be a subtype of the given type. A type assumption considers both the static and the dynamic type of the value of the expression to be the specified type for the purposes of the immediately enclosing function, method, or operator invocation or field access. This is in contrast to type ascription, which only gives a *static* type to an expression. Type assumption is used to access a method provided by a supertrait when multiple supertraits provide different methods with the same name. Fortress thus provides a richer version of type assumption operations such as `super` in the Java Programming Language.

13.31 Expression-like Functions

For convenience, the Fortress standard libraries provide functions such as `cast` and `instanceOf` that are often provided by other programming languages.

13.31.1 Casting

Although there is no “casting” operator (equivalent to casts in the Java Programming Language) built into Fortress, the effect of a cast can be provided by the following function:

```

cast[[T extends Any]](x: Any): T =
  typecase x of
    T => x
    else => throw CastException
  end

```

The function converts the type of its argument to a given type. If the static type of the argument is not a subtype of the given type, a `CastException` is thrown. For convenience, the function `cast` is included in the Fortress standard libraries.

13.31.2 Instanceof Testing

Although there is no “instanceof” operator (equivalent to instanceof testing in the Java Programming Language) built into Fortress, the effect of an instanceof testing can be provided by the following function:

```

instanceOf[[T extends Any]](x: Any): Boolean =
  typecase x of
    T => true
    else => false
  end

```

The function tests whether its argument has a given type and returns a boolean value. For convenience, the function `instanceOf` is included in the Fortress standard libraries.

13.31.3 Ignoring Values

For convenience, the function `ignore` (equivalent to the `ignore` function in the Objective Caml programming language) is included in the Fortress standard libraries:

```

ignore(x: Any) = ()

```

The function discards the value of its argument and returns `()`. For example, the following:

```

ignore(f x)

```

is equivalent to:

```

f x; ()

```

13.31.4 Enforcing Tuples

An identity function `tuple` is defined in the Fortress standard libraries to make “(*identifier* = *e*)” a tuple expression (as discussed in Section 13.27.6):

```

tuple[[T extends Tuple]](x: T) = x

```

The function returns its argument as a tuple expression.

13.31.5 Identity

An identity function `identity` is defined in the Fortress standard libraries:

$$\text{identity}[[T \text{ extends Any}]](x: T) = x$$

The function returns its argument as it is.

13.31.6 Coercion

An identity function *coerce* is defined in the Fortress standard libraries to convert the type of its argument to its type argument:

$$\text{coerce}[[T]](x: T) = x$$

The function returns its argument as the given type. Unlike coercions described in Chapter 17, the *coerce* function can apply to an argument whose type is a subtype of the type being coerced to.

Chapter 14

Exceptions

Exceptions are values that can be thrown and caught, via `throw` expressions (described in Section 13.24) and `catch` clauses of `try` expressions (described in Section 13.25). When a `throw` expression “`throw e`” is evaluated, the subexpression `e` is evaluated to an exception. The static type of `e` must be a subtype of `Exception`. Then the `throw` expression tries to transfer control to its *dynamically containing block* (described in Chapter 4), from the innermost outward, until either (i) an enclosing `try` expression is reached, with a `catch` clause matching a type of the thrown exception, or (ii) the outermost dynamically containing block is reached.

If a matching `catch` clause is reached, the right-hand side of the first matching subclass is evaluated. If no matching `catch` clause is found before the outermost dynamically containing block is reached, the outermost dynamically containing block completes abruptly whose cause is the thrown exception.

If an enclosing `try` expression of a `throw` expression includes a `finally` clause, and the `try` expression completes abruptly, the `finally` clause is evaluated before control is transferred to the dynamically containing block.

14.1 Causes of Exceptions

Every exception is thrown for one of the following reasons:

1. A `throw` expression is evaluated.
2. An implementation resource is exceeded (e.g., an attempt is made to allocate beyond the set of available locations).

14.2 Types of Exceptions

All exceptions have type `Exception` declared as follows:

```
trait Exception comprises { CheckedException, UncheckedException }
  settable message: Maybe[String]
  settable chain: Maybe[Exception]
  printStackTrace(): ()
end
```

Every exception has either type `CheckedException` or `UncheckedException`:

```

trait CheckedException
  extends { Exception }
  excludes { UncheckedException }
end

trait UncheckedException
  extends { Exception }
  excludes { CheckedException }
end

```

A functional declaration (described in Section 9.2 and Section 12.1) includes an optional `throws` clause in its header listing the `CheckedExceptions` (also written *checked exceptions*) that can be thrown by invocation of the functional. If a `throws` clause is not explicitly provided, the `throws` clause of the functional declaration is empty. The body of a functional is statically checked to ensure that no checked exceptions are thrown by any subexpression of the functional body other than those listed in the `throws` clause. This static check is performed by examining each `throw` expression and functional invocation I , determining the static type of the functional f invoked in I , and determining the `throws` clause of f . (If f is polymorphic, or occurs in a polymorphic context, instantiations of type variables free in the `throws` clause of f are substituted for formal type variables). For each checked exception thrown in I , the enclosing expressions of I are checked for a matching `catch` clause. The set A of all checked exceptions thrown by all invocations without a matching `catch` clause in the functional body is accumulated and compared against the `throws` clause of the enclosing functional declaration. If an exception that is not a subtype of an exception listed in the `throws` clause occurs in A , it is a static error.

A similar analysis is performed on top-level variable declarations. If it is determined that their initialization expressions can throw a checked exception, it is a static error.

14.3 Information of Exceptions

Every exception has optional fields: a message and a chained exception. These fields are default to `Nothing` as follows:

```

trait Exception comprises { CheckedException, UncheckedException }
  getter message(): Maybe[String] = Nothing
  setter message(String): ()
  getter chain(): Maybe[Exception] = Nothing
  setter chain(Exception): ()
  printStackTrace(): ()
end

```

where an optional value v is either `Nothing` or `Just(v)` as declared in Section 31.2. The `chain` field can be set at most once. If it is set more than once, an `InvalidChainException` is thrown. It is generally set when the exception is created.

When an exception is created, the execution stack of its thread at the time of the exception creation is captured in the exception. The invocation of `printStackTrace` prints the captured stack trace. There is no way to update the captured stack trace. If a programmer wants to catch a thrown exception and rethrow it, and capture the stack trace at the time of the second throwing of the exception, the programmer has to create a new exception (perhaps with the original exception as its `chain` field).

When an exception is thrown, its `message` and `chain` fields may be set. For example, if a checked exception is caught in a `catch` clause, and the `catch` clause in turn throws an unchecked exception, the unchecked exception can be chained so that an examination of the unchecked exception reveals information about the original exception. For example:

```

read(fileName) =
  try
    readFile(fileName)
  catch e
    IOException => throw Error("This code can't handle IOExceptions",e)
  end

```

where the *message* and *chain* fields of Error are set to "This code can't handle IOExceptions" and IOException respectively.

By default, a forbid clause in a try expression throws a new ForbiddenException by *chaining* the exception thrown by the try block in the try expression that is a subtype of the exception type listed in the forbid clause. For example, the following *read* function:

```

read(fileName) =
  try
    readFile(fileName)
  forbid IOException
  end

```

is equivalent to:

```

read(fileName) =
  try
    readFile(fileName)
  catch e
    IOException => throw ForbiddenException(e)
  end

```

where the *chain* of ForbiddenException is set to IOException.

Chapter 15

Overloading and Multiple Dispatch

Fortress allows functions and methods (collectively called as *functionals*) to be *overloaded*; that is, there may be multiple declarations for the same functional name visible in a single scope (which may include inherited method declarations), and several of them may be applicable to any particular functional call.

Calls to overloaded functionals are resolved by determining the most-specific applicable declaration. Fortress provides overloading rules (described in Chapter 33) for functional declarations that ensure there exists a unique most specific declaration for every call. Thus, it is unambiguous which declaration should be applied at run time.

In this chapter, we describe how to determine which declarations are *applicable* to a particular functional call, and when several are applicable, how to select among them. Section 15.1 introduces some terminology and notation. In Section 15.2, we show how to determine which declarations are applicable to a *named functional call* (a function call described in Section 13.6 or a naked method invocation described in Section 13.5) when all declarations have only ordinary parameters (without varargs or keyword parameters). We discuss how to handle dotted method calls (described in Section 13.4) in Section 15.3, and declarations with varargs and keyword parameters in Section 15.4. Determining which declaration is applied, if several are applicable, is discussed in Section 15.5.

15.1 Terminology and Notation

When there are two or more function declarations of the same name within a single lexical scope, we say that the function name is *overloaded* within that lexical scope; we also say that each of the function declarations is overloaded, and that any pair of the function declarations are *mutually overloaded*. Top-level function declarations in a component are permitted to be overloaded with function declarations imported from APIs (using “`import functionName from APIName`”). Likewise, it is permitted to have two or more method declarations (declared or inherited) of the same method name within a single trait or object declaration or object expression; we say that the method name is *overloaded* within that trait or object declaration or object expression, and we also say that each of the method declarations is overloaded, and that any pair of the method declarations are *mutually overloaded*.

We assume throughout this chapter that all static variables have been instantiated or inferred. Although there may be multiple declarations with the same functional name, it is a static error for their static parameters to differ (up to α -equivalence), or for one declaration to have static parameters and another to not have them. Hence, static parameters do not enter into the determination of which declarations are applicable, so we ignore them for most of this chapter.

An operator method declaration whose name is one of the operator parameters (described in Section 11.5) of its enclosing trait or object may be overloaded with other operator declarations in the same component. Therefore, such an operator method declaration must satisfy the overloading rules (described in Chapter 33) with every operator declaration in the same component.

Recall from Chapter 8 that we write $T \preceq U$ when T is a subtype of U , and $T \prec U$ when $T \preceq U$ and $T \neq U$.

15.2 Applicability to Named Functional Calls

In this section, we show how to determine which declarations are applicable to a named functional call when all declarations have only ordinary parameters (i.e., neither varargs nor keyword parameters).

For the purpose of defining applicability, a named functional call can be characterized by the name of the functional and its argument type. Recall that a functional has a single parameter, which may be a tuple (a dotted method has a receiver as well). We abuse notation by using *static call* $f(A)$ to refer to a named functional call with name f and whose argument has static type A , and *dynamic call* $f(X)$ to refer to a named functional call with name f and whose argument, when evaluated, has dynamic type X . (Note that if the type system is sound—and we certainly hope that it is!—then $X \preceq A$ for all well-typed calls to f .) We use the term *call* $f(C)$ to refer to static and dynamic calls collectively.

We also use *function declaration* $f(P) : U$ to refer to a function declaration with function name f , parameter type P , and return type U .

For method declarations, we must take into account the self parameter, which we do as follows:

A *dotted method declaration* $P_0.f(P) : U$ is a dotted method declaration with name f , where P_0 is the trait or object type in which the declaration appears, P is the parameter type, and U is the return type. (Note that despite the suggestive notation, a dotted method declaration need not explicitly list its self parameter. If the self parameter is listed then it must occur before the name of the method.)

A *functional method declaration* $f(P) : U$ with *self parameter at i* is a functional method declaration with name f , with the parameter `self` in the i th position, parameter type P , and return type U . Note that the static type of the self parameter is the trait or object trait type in which the declaration $f(P) : U$ occurs. In the following, we will use P_i to refer to the i th element of P .

We elide the return type of a declaration, writing $f(P)$ and $P_0.f(P)$, when the return type is not relevant to the discussion.

A declaration $f(P)$ is *applicable* to a call $f(C)$ if the call is in the scope of the declaration and $C \preceq P$. (See Chapter 7 for the definition of scope.)

Note that a named functional call $f(C)$ may invoke a dotted method declaration if the declaration is provided by the trait or object enclosing the call. To account for this let C_0 be the trait or object declaration immediately enclosing the call. Then we rewrite a named functional call $f(C)$ to $C_0.f(C)$ if there are no declarations applicable to $f(C)$. We then use the rule for applicability to dotted method calls (described in Section 15.3) to determine which declarations are applicable to $C_0.f(C)$.

15.3 Applicability to Dotted Method Calls

Dotted method applications can be characterized similarly to named functional applications, except that, analogously to dotted method declarations, we use A_0 to denote the static type of the receiver object, and, as for named functional calls, A to denote the static type of the argument of a static dotted method call; we use X_0 and X similarly for dynamic dotted method calls. We write $A_0.f(A)$ and $X_0.f(X)$ to refer to the static and dynamic calls respectively. A dotted method call $C_0.f(C)$ refers to static and dynamic calls collectively.

A dotted method declaration $P_0.f(P)$ is *applicable* to a dotted method call $C_0.f(C)$ if $C_0 \preceq P_0$ and $C \preceq P$.

15.4 Applicability for Functionals with Varargs and Keyword Parameters

The basic idea for handling varargs and keyword parameters is that we can think of a functional declaration that has such parameters as though it were (possibly infinitely) many declarations, one for each set of arguments it may be called with. In other words, we expand these declarations so that there exists a declaration for each number of arguments that can be passed to it.

A declaration with k keyword parameters corresponds to 2^k declarations which cannot be called with elided arguments, one for each subset of the keyword parameters. For example, the following declaration:

$$f(x = 5, y = 6, z = 7) : \mathbb{Z}$$

would be expanded into:

$$\begin{aligned} f(x = 5, y = 6, z = 7) &: \mathbb{Z} \\ f(x = 5, y = 6) &: \mathbb{Z} \\ f(x = 5, z = 7) &: \mathbb{Z} \\ f(y = 6, z = 7) &: \mathbb{Z} \\ f(x = 5) &: \mathbb{Z} \\ f(y = 6) &: \mathbb{Z} \\ f(z = 7) &: \mathbb{Z} \\ f() &: \mathbb{Z} \end{aligned}$$

Note that even though expanded declarations still have keyword parameters, they cannot be called with elided arguments any more. A declaration with keyword parameters is applicable to a call if any one of the expanded declarations is applicable.

A declaration with a varargs parameter corresponds to an infinite number of declarations, one for every number of arguments that may be passed to the varargs parameter. In practice, we can bound that number by the maximum number of arguments that the functional is called with anywhere in the program (in other words, a given program will contain only a finite number of calls with different numbers of arguments). The expansion described here is a conceptual one to simplify the description of the semantics; we do not expect a real implementation to actually expand these declarations at compile time. For example, the following declaration:

$$f(x : \mathbb{Z}, y : \mathbb{Z}, z : \mathbb{Z} \dots) : \mathbb{Z}$$

would be expanded into:

$$\begin{aligned} f(x : \mathbb{Z}, y : \mathbb{Z}, z : \mathbb{Z} \dots) &: \mathbb{Z} \\ f(x : \mathbb{Z}, y : \mathbb{Z}) &: \mathbb{Z} \\ f(x : \mathbb{Z}, y : \mathbb{Z}, z_1 : \mathbb{Z}) &: \mathbb{Z} \\ f(x : \mathbb{Z}, y : \mathbb{Z}, z_1 : \mathbb{Z}, z_2 : \mathbb{Z}) &: \mathbb{Z} \\ f(x : \mathbb{Z}, y : \mathbb{Z}, z_1 : \mathbb{Z}, z_2 : \mathbb{Z}, z_3 : \mathbb{Z}) &: \mathbb{Z} \\ \dots & \end{aligned}$$

Notice that the expansion includes the original declaration. This declaration is retained to account for the case when a tuple expression with a varargs expression is passed as an argument to a call; even though this declaration still has a varargs parameter, it's called with a fixed number of arguments. A declaration with a varargs parameter is applicable to a call if any one of the expanded declarations is applicable.

15.5 Overloading Resolution

Several declarations may be applicable to a given functional call. Therefore, it is necessary to determine which declaration is dispatched to. The basic principle is that, for any functional call, we wish to identify a unique declaration

that is the most specific among all declarations applicable to the call at run time. If there is no such declaration, then the call is *undefined*, which is a static error. If there are two or more such declarations, no one of which is more specific than all the others, the call is said to be *ambiguous*, which is also a static error. As discussed in Chapter 33, it is a static error for overloaded declarations to admit ambiguous calls at run time, whether such calls actually appear in the program or not.

If several declarations are applicable to a particular call, we determine which is most specific by using the subtype relation to compare parameter types. Formally, a declaration $f(P)$ is more specific than a declaration $f(Q)$ if $P \prec Q$. Similarly, a declaration $P_0.f(P)$ is more specific than a declaration $Q_0.f(Q)$ if $(P_0, P) \prec (Q_0, Q)$.

Chapter 16

Operators

Operators are like functions or methods; operator declarations are described in Chapter 34 and operator applications are described in Section 13.8. Just as functions or methods may be overloaded (see Chapter 15 for a discussion of overloading), so operators may have overloaded declarations, of the same or differing fixities. Calls to overloaded operators are resolved first via the fixity of the operators based on the context of the calls. Then, among the applicable declarations with that fixity, the most specific declaration is chosen.

Most operators can be used as prefix, infix, postfix, or nofix operators as described in Section 16.3; the fixity of an operator is determined syntactically, and the same operator may have declarations for multiple fixities. A simple example is that ‘`-`’ may be either infix or prefix, as is conventional. As another example, the Fortress standard libraries define ‘`!`’ to be a postfix operator that computes factorial when applied to integers. These operators may not be used as enclosing operators.

Several pairs of operators can be used as enclosing operators. Any number of ‘`|`’ (vertical line) can be used as both infix operators and enclosing operators.

Some operators are always postfix: a ‘`^`’ followed by any ordinary operator (with no intervening whitespace) is considered to be a superscripted postfix operator. For example, ‘`^*`’ and ‘`^+`’ and ‘`^?`’ are available for use as part of the syntax of extended regular expressions. As a very special case, ‘`^T`’ is also considered to be a superscripted postfix operator, typically used to signify matrix transposition.

Finally, there are special operators such as juxtaposition and operators on dimensions and units. Juxtaposition may be a function application or an infix operator in Fortress. When the left-hand-side expression is a function, juxtaposition performs function application; when the left-hand-side expression is a number, juxtaposition conventionally performs multiplication; when the left-hand-side expression is a string, juxtaposition conventionally performs string concatenation. Fortress provides several operators on dimensions and units as described in Chapter 18.

16.1 Operator Names

To support a rich mathematical notation, Fortress allows most Unicode characters that are specified to be mathematical operators to be used as operators in Fortress expressions, as well as these characters and character combinations:

```
!   @   #   $   %   *   +   -   =   |   :   <   >   /   ?   ^   ~  
->  --> =>  ==> <=  >=  =/=  **  !!  ||  |||  
<<  <<<  >>  >>> <-> <-/-  -/-> <=>  ===
```

In addition, a token that is made up of a mixture of uppercase letters and underscores (but no digits), does not begin or end with an underscore, and contains at least two different letters is also considered to be an operator:

MAX MIN $\sqrt{\quad}$ \times TIMES

The above operators are rendered as: MAX MIN $\sqrt{\quad}$ \times . Some of these uppercase tokens are considered to be equivalent to single Unicode characters, but even those that are not can still be used as operators. (See Appendix F for a detailed description of operator names in Fortress.)

16.2 Operator Precedence

Fortress specifies that certain operators have higher precedence than certain other operators, so that one need not use parentheses in all cases where operators are mixed in an expression. (See Appendix F for a detailed description of operator precedence in Fortress.) However, Fortress does not follow the practice of other programming languages in simply assigning an integer to each operator and then saying that the precedence of any two operators can be compared by comparing their assigned integers. Instead, Fortress relies on defining traditional groups of operators based on their meaning and shape, and specifies specific precedence relationships between some of these groups. If there is no specific precedence relationship between two operators, then parentheses must be used. For example, Fortress does not accept the expression $a + b \cup c$; one must write either $(a + b) \cup c$ or $a + (b \cup c)$. (Whether or not the result then makes any sense depends on what definitions have been made for the $+$ and \cup operators—see Chapter 34.)

Here are the basic principles of operator precedence in Fortress:

- Member selection ($.$) and method invocation ($.name(\dots)$) are not operators. They have higher precedence than any operator listed below.
- Subscripting ($[\]$), superscripting ($^$), and postfix operators have higher precedence than any operator listed below; within this group, these operations are left-associative (performed left-to-right).
- *Tight juxtaposition*, that is, juxtaposition without intervening whitespace, has higher precedence than any operator listed below. The associativity of tight juxtaposition is type-dependent; see Section 16.7.
- Next, *tight fractions*, that is, the use of the operator $/$ with no whitespace on either side, have higher precedence than any operator listed below. The tight-fraction operator has no precedence compared with itself, so it is not permitted to be used more than once in a tight fraction without use of parentheses.
- *Loose juxtaposition*, that is, juxtaposition with intervening whitespace, has higher precedence than any operator listed below. The associativity of loose juxtaposition is type-dependent and is different from that for tight juxtaposition; see Section 16.7. Note that *lopsided juxtaposition* (having whitespace on one side but not the other) is a static error as described in Section 16.3.
- Prefix operators have higher precedence than any operator listed below.
- The infix operators are partitioned into certain traditional groups, as explained below. They have higher precedence than any operator listed below.
- The equal symbol $=$ in binding context, the assignment operator $:=$, and compound assignment operators ($+=$, $-=$, $\wedge=$, $\vee=$, $\cap=$, $\cup=$, and so on as described in Section 13.8) have lower precedence than any operator listed above. Note that compound assignment operators themselves are not operator names.

The infix binary operators are divided into four general categories: arithmetic, relational, boolean, and other. The arithmetic operators are further categorized as multiplication/division/intersection, addition/subtraction/union, and other. The relational operators are further categorized as equivalence, inequivalence, chaining, and other. The boolean operators are further categorized as conjunctive, disjunctive, and other.

The arithmetic and relational operators are further divided into groups based on shape:

- “ordinary” operators: $+ - \cdot \times / \pm \mp \oplus \ominus \otimes \oslash \boxplus \boxminus \langle \lesseqgtr \langle \ll \gg \rangle \gg \gg \gg \gg \gg \gg \gg \gg$ etc.

The arithmetic operations in this group are further subdivided into “plain” ($+ - \cdot \times / \pm \mp$ etc.), “circled” ($\oplus \ominus \otimes \oslash$ etc.), “boxed” ($\boxplus \boxminus$ etc.), and so on; any of these groups may be used with the plain relational operators ($\langle \lesseqgtr \langle \ll \gg \rangle \gg \gg \gg \gg \gg \gg \gg \gg$ etc.), but the groups may not be mixed.

- “rounded horseshoe” or “set” operators: $\cap \cup \sqcap \sqcup \subset \supset \subseteq \supseteq \not\subset \not\supset \not\subseteq \not\supseteq$ etc.
- “square horseshoe” operators: $\sqcap \sqcup \sqsubset \sqsupset \sqsubseteq \sqsupseteq$ etc.
- “curly” operators: $\wedge \vee \prec \succ \gt \lt \not\prec \not\succ \not\gt \not\lt$ etc.
- “triangular” relations: $\triangleleft \trianglelefteq \triangleright \trianglerighteq \not\triangleleft \not\trianglelefteq \not\triangleright \not\trianglerighteq$ etc.
- “chickenfoot” relations: \leftrightarrow etc.

The principles of precedence for binary operators are then as follows:

- A multiplication or division or intersection operator has higher precedence than any addition or subtraction or union operator that is in the same shape group.
- Certain addition and subtraction operators come in pairs, such as $+$ and $-$, or \oplus and \ominus , which are considered to have the same precedence and so may be mixed within an expression and are grouped left-associatively. These addition-subtraction pairs are the *only* cases where two different operators are considered to have the same precedence.
- An arithmetic operator has higher precedence than any equivalence or inequivalence operator.
- An arithmetic operator has higher precedence than any relational operator that is in the same shape group.
- A relational operator has higher precedence than any boolean operator.
- A conjunctive boolean operator has higher precedence than any disjunctive boolean operator.

While the rules of precedence are complicated, they are intended to be both unsurprising and conservative. Note that operator precedence in Fortress is not always transitive; for example, while $+$ has higher precedence than $<$ (so you can write $a + b < c$ without parentheses), and $<$ has higher precedence than \vee (so you can write $a < b \vee c < d$ without parentheses), it is *not* true that $+$ has higher precedence than \vee —the expression $a \vee b + c$ is not permitted, and one must instead write $(a \vee b) + c$ or $a \vee (b + c)$.

Another point is that the various multiplication and division operators do *not* have “the same precedence”; they may not be mixed freely with each other. For example, one cannot write $u \cdot v \times w$; one must write $(u \cdot v) \times w$ or (more likely) $u \cdot (v \times w)$. Similarly, one cannot write $a \cdot b / c \cdot d$; but juxtaposition does bind more tightly than a loose (whitespace-surrounded) division slash, so one is allowed to write $a b / c d$, and this means the same as $(a b)/(c d)$. On the other hand, loose juxtaposition binds less tightly than a tight division slash, so that $a b/c d$ means the same as $a (b/c) d$. On the other other hand, tight juxtaposition binds more tightly than tight division, so that $(n + 1)/(n + 2)(n + 3)$ means the same as $(n + 1)/((n + 2)(n + 3))$.

There are two additional rules intended to catch misleading code: it is a static error for an operand of a tight infix or prefix operator to be a loose juxtaposition, and it is a static error if the rules of precedence determine that a use of infix operator a has higher precedence than a use of infix operator b , but that particular use of a is loose and that particular use of b is tight. Thus, for example, the expression $\sin x + y$ is permitted, but $\sin x + y$ is not permitted. Similarly, the expression $a \cdot b + c$ is permitted, as are $a \cdot b + c$ and $a \cdot b + c$, but $a \cdot b + c$ is not permitted. (The rule detects only the presence or absence of whitespace, not the amount of whitespace, so $a \cdot b + c$ is permitted. You have to draw the line somewhere.)

When in doubt, just use parentheses. If there’s a problem, the compiler will (probably) let you know.

16.3 Operator Fixity

Most operators in Fortress can be used variously as prefix, postfix, infix, nofix, or multifix operators. (See Section 16.4 for a discussion of how infix operators may be chained or treated as multifix operators.) Some operators can be used in pairs as enclosing (bracketing) operators—see Section 16.5. The Fortress language dictates only the rules of syntax; whether an operator has a meaning when used in a particular way depends only on whether there is a definition in the program for that operator when used in that particular way (see Chapter 34).

The fixity of a non-enclosing operator is determined by context. To the left of such an operator we may find (1) a *primary* expression (described below), (2) another operator, or (3) a comma, semicolon, or left encloser. To the right we may find (1) a primary expression, (2) another operator, (3) a comma, semicolon, or right encloser, or (4) a line break. A primary expression is an identifier, a literal, an expression enclosed by matching enclosers, a field selection, or an expression followed by a postfix operator. Considered in all combinations, this makes twelve possibilities. In some cases one must also consider whether or not whitespace separates the operator from what lies on either side. The rules of operator fixity are specified by Figure 16.1, where the center column indicates the fixity that results from the left and right context specified by the other columns.

left context	whitespace	operator fixity	whitespace	right context
primary	yes	infix	yes	primary
	yes	error (infix)	no	
	no	postfix	yes	
	no	infix	no	
primary	yes	infix	yes	operator
	yes	error (infix)	no	
	no	postfix	yes	
	no	infix	no	
primary	yes	error (postfix)		, ; right encloser
primary	yes	infix		line break
	no	postfix		
operator		prefix		primary
operator		prefix		operator
operator		error (nofix)		, ; right encloser
operator		error (nofix)		line break
, ; left encloser		prefix		primary
, ; left encloser		prefix		operator
, ; left encloser		nofix		, ; right encloser
, ; left encloser		error (prefix)		line break

Figure 16.1: Operator Fixity (I)

A case described in the center column of the table as an **error** is a static error; for such cases, the fixity mentioned in parentheses is the recommended treatment of the operator for the purpose of attempting to continue the parse in search of other errors.

The table may seem complicated, but it all boils down to a couple of practical rules of thumb:

1. Any operator can be prefix, postfix, infix, or nofix.
2. An infix operator can be *loose* (having whitespace on both sides) or *tight* (having whitespace on neither side), but it mustn't be *lopsided* (having whitespace on one side but not the other).
3. A postfix operator should have no whitespace before it and should be followed (possibly after some whitespace) by a comma, semicolon, right encloser, or line break.

left context	whitespace	operator fixity	whitespace	right context
primary	yes	infix	yes	primary
	yes	left encloser	no	
	no	right encloser	yes	
	no	infix	no	
primary	yes	infix	yes	operator
	yes	left encloser	no	
	no	right encloser	yes	
	no	infix	no	
primary	yes	error (right encloser)		, ; right encloser
	no	right encloser		
primary	yes	infix		line break
	no	right encloser		
operator		error (left encloser)	yes	primary
		left encloser	no	
operator		error (left encloser)	yes	operator
		left encloser	no	
operator		error (nofix)		, ; right encloser
operator		error (nofix)		line break
, ; left encloser		left encloser		primary
, ; left encloser		left encloser		operator
, ; left encloser		nofix		, ; right encloser
, ; left encloser		error (left encloser)		line break

Figure 16.2: Operator Fixity (II)

16.4 Chained and Multifix Operators

Certain infix mathematical operators that are traditionally regarded as *relational* operators, delivering boolean results, may be *chained*. For example, an expression such as $A \subseteq B \subset C \subseteq D$ is treated as being equivalent to $(A \subseteq B) \wedge (B \subset C) \wedge (C \subseteq D)$ except that the expressions B and C are evaluated only once (which matters only if they have side effects such as writes or input/output actions). Fortress restricts such chaining to operators of the same kind and having the same sense of monotonicity; for example, neither $A \subseteq B \leq C$ nor $A \subseteq B \supset C$ is permitted. Equivalence operators may be mixed into a chain; for example, one may write $A \subseteq B = C \subseteq D$. This transformation is done before type checking. In particular, it is done even though these operators do not return boolean values, and the resulting expression is checked for type correctness. (See Section F.4 for a detailed description of which operators may be chained.)

Any infix operator that does not chain may be treated as *multifix*. If $n - 1$ occurrences of the same operator separate n operands where $n \geq 3$, then the compiler first checks to see whether there is a definition for that operator that will accept n arguments. If so, that definition is used; if not, then the operator is treated as left-associative and the compiler looks for a two-argument definition for the operator to use for each occurrence. As an example, the cartesian product $S_1 \times S_2 \times \cdots \times S_n$ of n sets may usefully be defined as a multifix operator, but ordinary addition $p + q + r + s$ is normally treated as $((p + q) + r) + s$.

16.5 Enclosing Operators

These operators are always used in pairs as enclosing operators:

(/ /) (\ \)

[]	[/ /]	[* *]
{ }	{/ /}	{* *}
	</ />	<\ \>
	<</ />>	<<\ \>>

(ASCII encodings are shown here; they all correspond to particular single Unicode characters.) There are other pairs as well, such as [] and []. Note that the pairs () and [\ \] (also known as []) are not operators; they play special roles in the syntax of Fortress, and their behavior cannot be redefined by a library. The bracket pairs that may be used as enclosing operators are described in Section F.1.

Any number of ‘|’ (vertical line) may also be used in pairs as enclosing operators but there is a trick to it, because on the face of it you can’t tell whether any given occurrence is a left encloser or a right encloser. Again, context is used to decide, this time according to Figure 16.2.

This is very similar to Figure 16.1 in Section 16.3; a rough rule of thumb is that if an ordinary operator would be considered a prefix operator, then one of these will be considered a left encloser; and if an ordinary operator would be considered a postfix operator, then one of these will be considered a right encloser.

In this manner, one may use |...| for absolute values and ||...|| for matrix norms.

16.6 Conditional Operators

If a binary operator other than ‘:’ is immediately followed by a ‘:’ then it is *conditional*: evaluation of the right-hand operand cannot begin until evaluation of the left-hand operand has completed, and whether or not the right-hand operand is evaluated may depend on the value of the left-hand operand. If the left-hand operand throws an exception, then the right-hand operand is not evaluated.

The Fortress standard libraries define two conditional operators on boolean values, \wedge : and \vee : (see Section 16.8.14).

See Section 34.8 for a discussion of how conditional operators are implemented.

16.7 Juxtaposition

Juxtaposition in Fortress may be a function call or a special infix operator. See Section 25.1 for an example declaration of a `juxtaposition` operator.

When two expressions are juxtaposed, the juxtaposition is interpreted as follows: if the left-hand-side expression is a function, juxtaposition performs function application; otherwise, juxtaposition performs the `juxtaposition` operator application.

The manner in which a juxtaposition of three or more items should be associated requires type information and awareness of whitespace. (This is an inherent property of customary mathematical notation, which Fortress is designed to emulate where feasible.) Therefore a Fortress compiler must produce a provisional parse in which such multi-element juxtapositions are held in abeyance, then perform a type analysis on each element and use that information to rewrite the n-ary juxtaposition into a tree of binary juxtapositions.

All we need to know is whether the static type of each element of a juxtaposition is an arrow type. There are actually three legitimate possibilities for each element of a juxtaposition: (a) it has an arrow type, in which case it is considered to be a function element; (b) it has a type that is not an arrow type, in which case it is considered to be a non-function element; (c) it is an identifier that has no visible declaration, in which case it is considered to be a function element (and everything will work out okay if it turns out to be the name of an appropriate functional method).

The rules below are designed to forbid certain forms of notational ambiguity that can arise if the name of a functional method happens to be used also as the name of a variable. For example, suppose that trait T has a functional method of one parameter named n ; then in the code

```
do
  a: T = t
  n: ℤ = 14
  z = n a
end
```

it might not be clear whether the intended meaning was to invoke the functional method n on a or to multiply a by 14. The rules specify that such a situation is a static error.

The rules for reassociating a loose juxtaposition are as follows:

- First the loose juxtaposition is broken into nonempty chunks; wherever there is a non-function element followed by a function element, the latter begins a new chunk. Thus a chunk consists of some number (possibly zero) of functions followed by some number (possibly zero) of non-functions.
- It is a static error if any non-function element in a chunk is an unparenthesized identifier f and is followed by another non-function element whose type is such that f can be applied to that latter element as a functional method.
- The non-functions in each chunk, if any, are replaced by a single element consisting of the non-functions grouped left-associatively into binary juxtapositions.
- What remains in each chunk is then grouped right-associatively.
- It is a static error if an element of the original juxtaposition was the last element in its chunk before reassociation, the chunk was not the last chunk (and therefore the element in question is a non-function element), the element was an unparenthesized identifier f , and the type of the following chunk after reassociation is such that f can be applied to that following chunk as a functional method.
- Finally, the sequence of rewritten chunks is grouped left-associatively.

(Notice that no analysis of the types of newly constructed chunks is needed during this process.)

Here is an example: $n(n+1) \sin 3nx \log \log x$. Assuming that \sin and \log name functions in the usual manner and that n , $(n+1)$, and x are not functions, this loose juxtaposition splits into three chunks: $n(n+1)$ and $\sin 3nx$ and $\log \log x$. The first chunk has only two elements and needs no further reassociation. In the second chunk, the non-functions $3nx$ are replaced by $((3n)x)$. In the third chunk, there is only one non-function, so that remains unchanged; the chunk is the right-associated to form $(\log(\log x))$. Finally, the three chunks are left-associated, to produce the final interpretation $((n(n+1))(\sin((3n)x)))(\log(\log x))$. Now the original juxtaposition has been reduced to binary juxtaposition expressions.

The rules for reassociating a tight juxtaposition follow a different strategy:

- If the tight juxtaposition contains no function element, or if only the last element is a function, go on to the next step. Otherwise, consider the leftmost function element and examine the element that follows it. If that latter element is not parenthesized, it is a static error; otherwise, replace the two elements with a single element consisting of a new juxtaposition of the two elements (in the same order), and perform a type analysis on this new juxtaposition. (At this point, it is a static error if this new juxtaposition is preceded in the overall juxtaposition by a non-function element that is an unparenthesized identifier f , and the type of the new juxtaposition is such that f can be applied to it as a functional method.) Then repeat this step on the original juxtaposition (which is now one element shorter).
- The overall juxtaposition now either is a single element or consists entirely of non-function elements. It is a static error the overall juxtaposition now contains a non-function element that is an unparenthesized identifier f , and the type of the following element is such that f can be applied to it as a functional method.

- Left-associate the remaining elements of the juxtaposition.

(Note that this process requires type analysis of newly created chunks along the way.)

Here is an (admittedly contrived) example: $reduce(f)(a)(x+1)sqrt(x+2)$. Suppose that $reduce$ is a curried function that accepts a function f and returns a function that can be applied to an array a (the idea is to use the function f , which ought to take two arguments, to combine the elements of the array to produce an accumulated result).

The leftmost function is $reduce$, and the following element (f) is parenthesized, so the two elements are replaced with one: $(reduce(f))(a)(x+1)sqrt(x+2)$. Now type analysis determines that the element $(reduce(f))$ is a function.

The leftmost function is $(reduce(f))$, and the following element (a) is parenthesized, so the two elements are replaced with one: $((reduce(f))(a))(x+1)sqrt(x+2)$. Now type analysis determines that the element $((reduce(f))(a))$ is not a function.

The leftmost function is $(sqrt)$, and the following element $(x+2)$ is parenthesized, so the two elements are replaced with one: $((reduce(f))(a))(x+1)(sqrt(x+2))$. Now type analysis determines that the element $(sqrt(x+2))$ is not a function.

There are no functions remaining in the juxtaposition, so the remaining elements are left-associated:

$$(((reduce(f))(a))(x+1))(sqrt(x+2))$$

Now the original juxtaposition has been reduced to binary juxtaposition expressions.

16.8 Overview of Operators in the Fortress Standard Libraries

This section provides a high-level overview of the operators in the Fortress standard libraries. See Appendix F for the detailed rules for the operators provided by the Fortress standard libraries.

16.8.1 Prefix Operators

For all standard numeric types, the prefix operator $+$ simply returns its argument and the prefix operator $-$ returns the negative of its argument.

The operator \neg is the logical NOT operator on boolean values and boolean intervals.

The operator \neg computes the bitwise NOT of an integer.

Big operators such as \sum begin a *reduction expression* (Section 13.17). The big operators include \sum (summation) and \prod (product), along with \cap , \cup , \wedge , \vee , $\underline{\vee}$, \oplus , \otimes , \oplus , \boxplus , \boxtimes , \boxtimes , MAX, MIN, and so on.

16.8.2 Postfix Operators

The operator $!$ computes factorial; the operator $!!$ computes double factorials. They may be applied to a value of any integral type and produces a result of the same type.

When applied to a floating-point value x , $x!$ computes $\Gamma(1+x)$, where Γ is the Euler gamma function.

16.8.3 Enclosing Operators

When used as left and right enclosing operators, `| |` computes the absolute value or magnitude of any number, and is also used to compute the number of elements in an aggregate, for example the cardinality of a set or the length of a list. Similarly, `||` is used to compute the norm of a vector or matrix.

The floor operator `[]` and ceiling operator `⌈ ⌋` may be applied to any standard integer, rational, or real value (their behavior is trivial when applied to integers, of course). The operators hyperfloor `⌊x⌋ = 2⌊log2 x`, hyperceiling `⌈x⌈ = 2⌈log2 x`, hyperhyperfloor `⌘x⌘ = 2⌊log2 x`, and hyperhyperceiling `⌘x⌘ = 2⌈log2 x` are also available.

16.8.4 Exponentiation

Given two expressions e and e' denoting numeric quantities v and v' that are not vectors or matrices, the expression $e^{e'}$ denotes the quantity obtained by raising v to the power v' . This operation is defined in the usual way on numerals.

Given an expression e denoting a vector and an expression e' denoting a value of type \mathbb{Z} , the expression $e^{e'}$ denotes repeated vector multiplication of e by itself e' times.

Given an expression e denoting a square matrix and an expression e' denoting a value of type \mathbb{Z} , the expression $e^{e'}$ denotes repeated matrix multiplication of e by itself e' times.

16.8.5 Superscript Operators

The superscript operator `^T` transposes a matrix. It also converts a column vector to a row vector or a row vector to a column vector.

16.8.6 Subscript Operators

Subscripting of arrays and other aggregates is written using square brackets:

<code>a[i]</code>	<i>is displayed as</i>	a_i	i th element of one-dimensional array a
<code>m[i, j]</code>	<i>is displayed as</i>	m_{ij}	i, j th element of two-dimensional matrix m
<code>space[i, j, k]</code>	<i>is displayed as</i>	$space_{ijk}$	i, j, k th element of three-dimensional array $space$
<code>a[3] := 4</code>	<i>is displayed as</i>	$a_3 := 4$	assign 4 to the third element of mutable array a
<code>m["foo"]</code>	<i>is displayed as</i>	m_{foo}	fetch the entry associated with string “foo” from map m

16.8.7 Multiplication, Division, Modulo, and Remainder Operators

For most integer, rational, floating-point, complex, and interval expressions, multiplication can be expressed using any of `'·'` or `'×'` or simply juxtaposition. There are, however, two subtle points to watch out for. First, juxtaposition of numerals is treated as literal concatenation rather than multiplication; in this way one can use spaces to separate groups of digits, for example `1 234 567.890 12` rather than `123457.89012`. Second, the `×` operator is used to express the shape of matrices, so if expressions using multiplication are used in expressing the shape of a matrix, it may be necessary to avoid the use of `'×'` to express multiplication, or to use parentheses.

For integer, rational, floating-point, complex, and interval expressions, division is expressed by `/`. When the operator `/` is used to divide one integer by another, the result is rational. The operator `÷` performs truncating integer division: $m \div n = \text{signum}(\frac{m}{n}) \lfloor \frac{m}{n} \rfloor$. The operator `REM` gives the remainder from such a truncating division: $m \text{ REM } n = m - n(m \div n)$. The operator `MOD` gives the remainder from a floor division: $m \text{ MOD } n = m - n \lfloor \frac{m}{n} \rfloor$;

when $n > 0$ this is the usual modulus computation that evaluates integer m to an integer k such that $0 \leq k < n$ and n evenly divides $m - k$.

The special operators `DIVREM` and `DIVMOD` each return a pair of values, the quotient and the remainder; $m \text{ DIVREM } n$ returns $(m \div n, m \text{ REM } n)$ while $m \text{ DIVMOD } n$ returns $(\lfloor \frac{m}{n} \rfloor, m \text{ MOD } n)$.

Multiplication of a vector or matrix by a scalar is done with juxtaposition, as is multiplication of a vector by a matrix (on either side). Vector dot product is expressed by ‘ \cdot ’ and vector cross product by ‘ \times ’. Division of a matrix or vector by a scalar may be expressed using ‘ $/$ ’.

The syntactic interaction of juxtaposition, \cdot , \times , and $/$ is subtle. See Section 16.2 for a discussion of the relative precedence of these operations and how precedence may depend on the use of whitespace.

The handling of overflow depends on the type of the number produced. For integer results, overflow throws an `IntegerOverflowException`. Rational computations do not overflow. For floating-point results, overflow produces $+\infty$ or $-\infty$ according to the rules of IEEE 754. For intervals, overflow produces an appropriate containing interval.

Underflow is relevant only to floating-point computations and is handled according to the rules of IEEE 754.

The handling of division by zero depends on the type of the number produced. For integer results, division by zero throws a `DivideByZeroException`. For rational results, division by zero produces $1/0$. For floating-point results, division by zero produces a NaN value according to the rules of IEEE 754. For intervals, division by zero produces an appropriate containing interval (which under many circumstances will be the interval of all possible real values and infinities).

Wraparound multiplication on fixed-size integers is expressed by $\dot{\times}$. Saturating multiplication on fixed-size integers is expressed by \boxtimes or \boxtimes . These operations do not overflow.

Ordinary multiplication and division of floating-point numbers always use the IEEE 754 “round to nearest” rounding mode. This rounding mode may be emphasized by using the operators \otimes (or \odot) and \oslash . Multiplication and division in “round toward zero” mode may be expressed with \boxtimes (or \boxtimes) and \boxdiv . Multiplication and division in “round toward positive infinity” mode may be expressed with \blacktriangle (or \blacktriangle) and \blacktriangleright . Multiplication and division in “round toward negative infinity” mode may be expressed with \blacktriangledown (or \blacktriangledown) and \blacktriangleright .

16.8.8 Addition and Subtraction Operators

Addition and subtraction are expressed with $+$ and $-$ on all numeric quantities, including intervals, as well as vectors and matrices.

The handling of overflow depends on the type of the number produced. For integer results, overflow throws an `IntegerOverflowException`. Rational computations do not overflow. For floating-point results, overflow produces $+\infty$ or $-\infty$ according to the rules of IEEE 754. For intervals, overflow produces an appropriate containing interval.

Underflow is relevant only to floating-point computations and is handled according to the rules of IEEE 754.

Wraparound addition and subtraction on fixed-size integers are expressed by $\dot{+}$ and $\dot{-}$. Saturating addition and subtraction on fixed-size integers are expressed by \boxplus and \boxminus . These operations do not overflow.

Ordinary addition and subtraction of floating-point numbers always use the IEEE 754 “round to nearest” rounding mode. This rounding mode may be emphasized by using the operators \oplus and \ominus . Addition and subtraction in “round toward zero” mode may be expressed with \boxplus and \boxminus . Addition and subtraction in “round toward positive infinity” mode may be expressed with \blacktriangle and \blacktriangle . Addition and subtraction in “round toward negative infinity” mode may be expressed with \blacktriangledown and \blacktriangledown .

The construction $x \pm y$ produces the interval $\langle x - y, x + y \rangle$.

16.8.9 Intersection, Union, and Set Difference Operators

Sets support the operations of intersection \cap , union \cup , disjoint union \uplus , set difference \setminus , and symmetric set difference \ominus . Disjoint union throws `DisjointUnionException` if the arguments are in fact not disjoint.

Intervals support the operations of intersection \cap , union \cup , and interior hull $\underline{\cup}$. The operation \cap returns a pair of intervals; if the intersection of the arguments is a single contiguous span of real numbers, then the first result is an interval representing that span and the second result is an empty interval, but if the intersection is two spans, then two (disjoint) intervals are returned. The operation \cup returns a pair of intervals; if the arguments overlap, then the first result is the union of the two intervals and the second result is an empty interval, but if the arguments are disjoint, they are simply returned as is.

16.8.10 Minimum and Maximum Operators

The operator `MAX` returns the larger of its two operands, and `MIN` returns the smaller of its two operands.

For floating-point numbers, if either argument is a NaN then NaN is returned. The floating-point operations `MAXNUM` and `MINNUM` behave similarly except that if one argument is NaN and the other is a number, the number is returned. For all four of these operators, when applied to floating-point values, -0 is considered to be smaller than $+0$.

16.8.11 GCD, LCM, and CHOOSE Operators

The infix operator `GCD` computes the greatest common divisor of its integer operands, and `LCM` computes the least common multiple. The operator `CHOOSE` computes binomial coefficients: $n \text{ CHOOSE } k = \binom{n}{k} = \frac{n!}{k!(n-k)!}$.

The expression $e \neq e'$ is semantically equivalent to the expression $\neg(e = e')$.

16.8.12 Comparisons Operators

Unless otherwise noted, the operators described in this section produce boolean (*true/false*) results.

The operators $<$, \leq , \geq , and $>$ are used for numerical comparisons and are supported by integer, rational, and floating-point types. Comparison of rational values throws `RationalComparisonException` if either argument is the rational infinity $1/0$ or the rational indefinite $0/0$. Comparison of floating-point values throws `FloatingComparisonException` if either argument is a NaN.

The operators $<$, \leq , \geq , and $>$ may also be used to compare characters (according to the numerical value of their Unicode codepoint values) and strings (lexicographic order based on the character ordering). They also use lexicographic order when used to compare lists whose elements support these same comparison operators.

When $<$, \leq , \geq , and $>$ are used to compare numerical intervals, the result is a boolean interval. The functions *possibly* and *certainly* are useful for converting boolean intervals to boolean values for testing. Thus *possibly*($x > y$) is true if and only if there is some value in the interval x that is greater than some value in the interval y , while *certainly*($x > y$) is true if and only if x and y are nonempty and every value in x is greater than every value in y .

The operators \subset , \subseteq , \supseteq , and \supset may be used to compare sets or intervals regarded as sets.

The operator \in may be used to test whether a value is a member of a set, list, array, interval, or range.

16.8.13 Logical Operators

The following binary operators may be used on boolean values:

\wedge	AND
\vee	inclusive OR
$\underline{\vee}$ or \oplus or \neq	exclusive OR
\equiv or \leftrightarrow or $=$	equivalence (if and only if)
\rightarrow	IMPLIES
$\overline{\wedge}$	NAND
$\overline{\vee}$	NOR

These same operators may also be applied to boolean intervals to produce boolean interval results.

The following operators may be used on integers to perform “bitwise” operations:

$\&$	bitwise AND
$\& \vee$	bitwise inclusive OR
$\& \underline{\vee}$	bitwise exclusive OR

The prefix operator \neg computes the bitwise NOT of an integer.

16.8.14 Conditional Operators

If $p(x)$ and $q(x)$ are expressions that produce boolean results, the expression $p(x) \wedge : q(x)$ computes the logical AND of those two results by first evaluating $p(x)$. If the result of $p(x)$ is *true*, then $q(x)$ is also evaluated, and its result becomes the result of the entire expression; but if the result of $p(x)$ is *false*, then $q(x)$ is not evaluated, and the result of the entire expression is *false* without further ado. (Similarly, evaluating the expression $p(x) \vee : q(x)$ does not evaluate $q(x)$ if the result of $p(x)$ is *true*.) Contrast this with the expression $p(x) \wedge q(x)$ (with no colon), which evaluates both $p(x)$ and $q(x)$, in no specified order and possibly in parallel.

Chapter 17

Conversions and Coercions

Fortress provides a mechanism, called *coercion*, to allow a value of one type to be automatically converted to a value of another type. Programmers can define coercions (described in Section 17.2) and coercions may change the set of applicable declarations for a call (as described in Section 17.4). If multiple coercions can be applied to a particular functional call then the most appropriate coercion is chosen statically. Restrictions on coercion declarations (described in Section 17.6) guarantee that the static resolution of coercion (described in Section 17.5) is well-defined. Fortress provides implicit coercions for tuple types and arrow types (described in Section 17.7). Fortress also provides an additional feature of coercion that allows “widest-need” evaluation of numerical expressions (described in Section 17.8).

17.1 Principles of Coercion

In certain situations it is convenient to be able to use a value of one type T as if it were a value of another type U even though T is not a subtype of U . For example, it is convenient to be able to use the integer-valued expression 2 in a floating-point expression even though its type is not a subtype of any floating-point type. Fortress supports the automatic conversion of integer values to floating-point values (the technical term for this kind of automatic conversion is coercion). In this way one can write $(x + 1)/2$ rather than $(x + 1.0)/2.0$, for example. Such coercion applies generally to function and method (collectively called as functional) calls as well as to operators; one can write $\ln 2$ rather than $\ln 2.0$, or $\arctan(1, 1)$ rather than $\arctan(1.0, 1.0)$, for example.

One way to think about coercion is that if type T can be coerced to type U , then a value of type T can be used to “stand in” for a value of type U in a functional call, for example. This is different from actually *being* of type U ; it means only that, given any value of type T , an appropriate value of type U can be computed to substitute for it. Also, coercion occurs only when the declared type of the corresponding parameter in the functional declaration is exactly the type U being coerced to not if it is a supertype of U . Note, however, that if type T can be coerced to type U then any subtype of T can also be coerced to type U .

Coercion from T to U may occur in an expression context where the context expects the expression to have type U and the type of the expression is a subtype of T except for the following cases:

- type ascriptions (see Section 13.29)
- type assumptions (see Section 13.30)
- arguments to an explicit coercion invocation (see Section 17.3)

For example, a coercion may occur in a variable definition when the declared type of the variable is U and the type of the expression on the right-hand side is a subtype of T .

The expression contexts where a coercion can occur are:

- the right-hand sides of variable and field declarations where the left-hand sides have declared types
- arguments to functionals and constructors where the corresponding parameters have declared types
- body expressions of functionals and constructors where the return types are declared
- the right-hand sides of test and property declarations
- expressions in contracts
- expressions whose enclosing expressions constrain the types of their subexpressions

Chapter 13 describes the type constraints for each expression in Fortress.

Coercion is *not* automatically chained in Fortress, unlike some other programming languages: even if type T can be coerced to type U and type U can be coerced to type V , it is not necessarily the case that type T can be coerced to type V . Type T can be coerced to type V only if this coercion is explicitly defined.

Example 1: For any floating-point parameter, a decimal integer literal argument may be used.

Example 2: For any floating-point parameter, a floating-point argument of a shorter format may be used.

Example 3: For any floating-point interval parameter, a non-interval floating-point argument (of the same or shorter floating-point format) may be used.

17.2 Coercion Declarations

Syntax:

```
Coercion ::= widening? coercion StaticParams? (Id IsType) CoercionClauses = Expr
CoercionClauses ::= Throws? CoercionWhere? Contract
CoercionWhere ::= where { CoercionWhereClauseList }
CoercionWhereClauseList ::= CoercionWhereClause(, CoercionWhereClause)*
CoercionWhereClause ::= WhereClause
| TypeRef widens or coerces TypeRef
```

To declare that trait U allows a coercion from type T , the declaration of trait U must provide a coercion declaration whose parameter type is T . Coercion declarations are like functional declarations, except that `coercion` is actually a reserved word, a coercion declaration may have special `where`-clause constraints (described below), and it is not permitted to specify a return type, because the return type must be the very trait in whose declaration the coercion declaration appears. The coercion body is required; there is no such thing as an abstract coercion declaration.

Coercions may have static parameters (described in Chapter 11) and a `where` clause (described in Section 11.6), just like functionals. The `where` clause may contain one of the following special constraints:

```
Type1 coerces Type2
Type1 widens Type2
Type1 widens or coerces Type2
```

The first is true if trait $Type_1$ has a coercion from $Type_2$. The second is true if trait $Type_1$ has a widening coercion (described in Section 17.8) from $Type_2$. The third may be used only in a coercion with the “widening” reserved word; it is true if $Type_1$ has a coercion from $Type_2$, and furthermore the coercion declaration to which the `where` clause belongs is widening only if $Type_1$ has a widening coercion from $Type_2$. For example:

```

trait Vector[[T extends Number]]
  widening coercion [[U extends Number]](x: Vector[[U]])
  where { T widens or coerces U } = ...
end

```

Coercions, unlike methods, are not inherited. If trait V extends trait U , and trait U has a coercion from type T , then V does *not* thereby have a coercion from type T . (It may, however, have its own coercion from type T , separately defined within the body of V .) For example, given the following declarations,

```

trait A
  coercion (b: B) = ...
end
object B end
trait C extends A end
f(c: C) = 5

```

a call to $f(B)$ is considered a static error because trait C does not inherit a coercion from type B .

17.3 Coercion Invocations

One may invoke a coercion explicitly with the syntax: `Trait.coercion(expr)`. Overloading resolution (described in Section 15.5) applies in the usual way if the specified trait has more than one coercion that applies to the actual argument. Implicit coercions cannot occur in an argument to an explicit coercion invocation. Result of an explicit coercion can be implicitly coerced.

One way to think about coercion is that explicit calls to coercion declarations are automatically inserted into the expression in a context where a coercion can occur. For example, if trait U has a coercion from type T , then whenever a functional, f , is declared with a parameter of type U , a call $f(t)$ where t has type T can be rewritten to $f(U.coercion(t))$ making the declaration of f applicable to the call. However, this rewriting does not occur if there exists a declaration that is applicable to the call before the rewriting (see Section 17.5 for further discussion).

If coercion is possible for more than one element of a tuple argument, a cross-product effect is obtained. For example, in this code:

```

object X
  coercion (kiki: Y) = ...
  coercion (tutu: Q) = ...
  hack(other: X) = 1
end
object Y end
object Q end
foo(dodo: X) = 2
bar(hyar: X, yon: X) = 3
bar(hyar: Q, yon: Q) = 4

```

the following method invocations are valid:

```

X.hack(Y)
X.hack(Q)

```

Because there is no declaration of *hack* applicable to Y and Q , these invocations are rewritten to:

```

X.hack(X.coercion(Y))
X.hack(X.coercion(Q))

```

Similarly, the function calls in the left-hand side of the following are valid and rewritten to the right-hand side:

```

foo(Y)  is rewritten to  foo(X.coercion(Y))
foo(Q)  is rewritten to  foo(X.coercion(Q))

bar(Y,X) is rewritten to bar(X.coercion(Y),X)
bar(Q,X) is rewritten to bar(X.coercion(Q),X)
bar(X,Y) is rewritten to bar(X,X.coercion(Y))
bar(X,Q) is rewritten to bar(X,X.coercion(Q))
bar(Y,Y) is rewritten to bar(X.coercion(Y),X.coercion(Y))
bar(Q,Q) is rewritten to bar(Q,Q)
bar(Q,Y) is rewritten to bar(X.coercion(Q),X.coercion(Y))
bar(Y,Q) is rewritten to bar(X.coercion(Y),X.coercion(Q))

```

Note that the call `bar(Q, Q)` remains unchanged because there exists a declaration for `bar` that is applicable without coercion.

To continue the example, let us illustrate a point about type parameters. If we add the following definitions:

```

trait Frobbos
  frob(item : X) = 5
  coercion(m : Matrix) = ...(* irrelevant! *)
end

object Bozo[[T extends Frobbos]](t : T) ... end

```

then this says nothing about whether the type parameter `T` of `Bozo` has coercions, because coercions are not inherited. (In fact, we can make a stronger statement: types named by type parameters *do not have coercions!*) But it is guaranteed that the following method invocations are valid if they occur within the declaration of `Bozo`:

```

t.frob(X)
t.frob(Y)
t.frob(Q)

```

because `T` inherits the method declaration `frob(item : X)` from `Frobbos` and `X` contains coercions from `Y` and `Q`.

17.4 Applicability with Coercion

As discussed in the previous section, coercion in Fortress alters the set of applicable declarations for a particular functional call. In this section we formally define the applicability of declarations with coercion. (This level of formality is necessary to discuss the interaction of overloading and coercion.) We build on the terminology and notation defined in Chapter 15.

We write $T \rightarrow U$ if U defines a coercion from T . We say that T can be coerced to U , and we write $T \rightsquigarrow U$, if U defines a coercion from T or any supertype of T ; that is, $T \rightsquigarrow U \iff \exists T' : T \preceq T' \wedge T' \rightarrow U$.

Because of automatic coercion, a declaration may be applicable even if its parameter type is not a supertype of the argument type of the call. We define a new relation, *substitutability*, that takes coercion into account.

We say that a type T is *substitutable* for type U , and we write $T \sqsubseteq U$, if T is a subtype of U or T can be coerced to U ; that is,

$$T \sqsubseteq U \iff T \preceq U \vee T \rightsquigarrow U.$$

Note that \sqsubseteq need not be transitive. This is a result of the fact that coercion does not chain. However, if T is substitutable for U then so are T 's subtypes; that is, $A \preceq T \wedge T \sqsubseteq U \implies A \sqsubseteq U$.

A declaration $f(P)$ is *applicable with coercion* to a call $f(C)$ if the call is in the scope of the declaration and $C \sqsubseteq P$. (See Chapter 7 for the definition of scope.)

Recall that Section 15.2 describes a rewriting of named functional calls into dotted method calls when no declarations are applicable to the named functional call. This rewriting is also used to determine applicability with coercion. If there is no declaration applicable with coercion to a named functional call then the call is rewritten into a dotted method call (as described in Section 15.2). Applicable declarations are then determined by the rules for applicability with coercion for dotted method calls.

Similarly, a dotted method declaration $P_0.f(P)$ is *applicable with coercion* to a dotted method call $C_0.f(C)$ if it $C_0 \preceq P_0$ and $C \sqsubseteq P$. Notice that the receiver is compared using the subtype relation instead of the substitutable relation. This reflects the restriction that receivers of dotted methods cannot be coerced. However, `self` parameters of functional methods can be coerced. This distinction is consistent with the intuition that functional methods are rewritten into top-level functions.

Note that the only difference between applicability and applicability with coercion is that substitutability is used instead of subtyping to compare the parameter types of the declarations.

Also note that the definition of applicability with coercion does not take keyword parameters or varargs parameters into account. Such declarations are conceptually rewritten into numerous declarations which cannot be called with elided arguments nor with variable number of arguments (as described in Section 15.4). If one of the expanded declarations is applicable with coercion to a call, then the original declaration (with keyword or varargs parameters) is applicable with coercion to the call.

17.5 Coercion Resolution

For a given functional call, we first determine whether there exists a declaration that is applicable without coercion. If so, the most specific declaration is selected; if not, then coercions are explicitly added to the functional call. If more than one coercion is possible then the coercion that yields the most specific type is chosen. Section 17.6 and Chapter 33 describe overloading rules for the declarations of coercions and overloaded functionals that guarantee there exists always a most specific coercion when two or more are possible.

We first define a notion of more specific types in the presence of coercion. Recall from Section 8.1 that Fortress defines an *exclusion* relation between types which is denoted by \diamond .

For types T and U , we say that T *rejects* U , and write $T \rightarrow U$, if for all coercions to T , the type coerced from excludes U :

$$T \rightarrow U \iff \forall A : A \rightarrow T \implies A \diamond U.$$

Note that $T \rightarrow U$ implies $U \not\sim T$.

We say that T is *no less specific* than U , and write $T \trianglelefteq U$, if T is a subtype of U or T excludes, can be coerced to, and rejects U :

$$T \trianglelefteq U \iff T \preceq U \vee (T \diamond U \wedge T \rightsquigarrow U \wedge T \rightarrow U).$$

The \trianglelefteq relation is reflexive and antisymmetric but not necessarily transitive. It is possible, for example, for three types, T , U and V , each excluding the other two, to have coercions from T to U and from U to V . In this case, $T \trianglelefteq U$ and $U \trianglelefteq V$ but $T \not\trianglelefteq V$.

Notice that if two tuple types have a bijective correspondence between their element types then the \trianglelefteq relationship between the tuple types is equivalent to applying the \trianglelefteq relation elementwise.

We say that T is *more specific* than U , and write $T \triangleleft U$, if $T \trianglelefteq U \wedge T \neq U$.

We extend the definitions of no less specific and more specific to declarations. We say that a declaration $f(P)$ is *no less specific* than a declaration $f(Q)$ if $P \trianglelefteq Q$. Similarly, we say that a declaration $P_0.f(P)$ is *no less specific* than a declaration $Q_0.f(Q)$ if $(P_0, P) \trianglelefteq (Q_0, Q)$. A declaration $f(P)$ is *more specific* than a declaration $f(Q)$ if $P \triangleleft Q$. Similarly, a declaration $P_0.f(P)$ is *more specific* than a declaration $Q_0.f(Q)$ if $(P_0, P) \triangleleft (Q_0, Q)$.

Now we describe how to determine which coercion is applied to a given call in terms of rewriting functional calls. The rewriting is for pedagogical purposes; implementation techniques may vary.

Consider a static call $f(A)$ or $A_0.f(A)$. Let Σ be the set of parameter types of functional declarations of f that are applicable to the call. Let Σ' be the set of parameter types of functional declarations of f that are applicable with coercion to the call. If Σ is not empty then we use the overloading resolution described in Section 15.5 to determine which element of Σ is called. If Σ is empty but Σ' is not, then we rewrite the call as follows. Define:

$$C = \{T \in \Sigma' \mid S \rightarrow T \wedge A \trianglelefteq S\}.$$

Let $T \in C$ be the most specific element of C . In other words, there does not exist $T' \in C$ such that $T' \triangleleft T$. The restrictions given in Section 17.6 and Chapter 33 guarantee that such a T exists and that it is unique. The call $f(A)$ or $A_0.f(A)$ is rewritten to $f(T.\text{coercion}(A))$ or $A_0.f(T.\text{coercion}(A))$ and the declaration with parameter type T is applied to the call.

As an example, consider the following definitions:

```

trait Z32 end
trait Z64
  coercion (z:Z32) = ...
end
trait Z128
  coercion (z:Z32) = ...
  coercion (z:Z64) = ...
end
f(z:Z64) = 1
f(z:Z128) = 2

```

Then the call $f(\mathbb{Z}32)$ resolves to the declaration $f(\mathbb{Z}64)$ because $\mathbb{Z}64 \triangleleft \mathbb{Z}128$, which follows from the fact that $\mathbb{Z}64$ coerces to $\mathbb{Z}128$.

Notice that coercion is resolved statically. Once a coercion is statically chosen for a call, this coercion is conceptually inserted into the call. This means that the statically chosen coercion is applied at run time.

For example, in the following program:

```

trait A
  coercion (c:C) = ...
end
trait B excludes A end
trait C end
object D extends {B, C} end
f(a:A) = 3
f(b:B) = 4
c:C = D
f(c)

```

the call to $f(C)$ resolves to the declaration $f(A)$ despite the fact that the declaration $f(B)$ is applicable to the dynamic call $f(D)$ and does not require coercion.

17.6 Restrictions on Coercion Declarations

We place two restrictions on coercion declarations to ensure that it is always possible to resolve coercions.

It is a static error for a type to define a coercion from any of its subtypes. For example, the following program is statically rejected:

```
trait Number
  coercion (z:ℤ) = ...
end
trait ℤ extends Number end
```

Instead, z of type \mathbb{Z} can be coerced to type `Number` by “`coerce[[Number]](z)`” where `coerce` is defined in the Fortress standard libraries (as described in Section 13.31.6).

It is also a static error for cycles to exist in the type hierarchy produced by extension and coercion declarations. Such cycles may be composed of solely subtype relationships or solely coercion or a mixture of the two. In all cases the cycles are statically rejected. An example showing a cycle that is a mixture of subtype and coercion relationships follows:

```
trait A end
trait B extends A end
trait C extends B
  coercion (a:A) = ...
end
```

17.7 Coercions for Tuple and Arrow Types

Unlike other types, tuple types (described in Section 8.4) and arrow types (described in Section 8.5) are not defined explicitly. Therefore coercions to these types are also not defined explicitly. Instead, the following rules describe when these coercions are implicitly defined.

There is a coercion from a tuple type X to a tuple type Y if all the following conditions hold:

1. X is not a subtype of Y ;
2. for every plain type T in Y , there is a corresponding plain type in X whose type is substitutable for T ;
3. if neither X nor Y has a varargs type, then they have the same number of plain types;
4. if X has a varargs type, then Y has a varargs type, the type S of the varargs type $S\dots$ in X is substitutable for the type T of the varargs type $T\dots$ in Y , and X and Y have the same number of plain types;
5. if X has no varargs type and Y has a varargs type $T\dots$, then every plain type in X that has no corresponding plain type in Y is substitutable for T ; and
6. the correspondence between keyword-type pairs in X and Y is bijective, and the type of each such pair in X is substitutable for the type in the corresponding pair in Y .

Tuple type coercions are invoked by distributing the coercion elementwise. However, if an element type of the tuple type being coerced from is a subtype of the corresponding element type of the tuple type being coerced to, the coercion is ignored. For example, the following coercions:

$(A, B). \text{ coercion } (x, y)$
 $(\text{kw}d = A). \text{ coercion } (\text{kw}d = x)$

are rewritten to:

$(A. \text{ coercion } (x), B. \text{ coercion } (y))$
 $(\text{kw}d = A. \text{ coercion } (x))$

However, if the type of y were a subtype of B then the first coercion would instead be rewritten to:

$(A. \text{ coercion } (x), y)$

Coercions to varargs types such as the following:

$(A \dots). \text{ coercion } (x, y)$

are invoked by applying the coercion to the type in the varargs type, A in this example, to each of the elements of the tuple. Additionally, there is a coercion from any type T to a tuple type solely with varargs type $(T \dots)$.

There is a coercion from arrow type “ $A \rightarrow B \text{ throws } C$ ” to arrow type “ $D \rightarrow E \text{ throws } F$ ” if all of the following conditions hold:

1. “ $A \rightarrow B \text{ throws } C$ ” is not a subtype of “ $D \rightarrow E \text{ throws } F$ ”;
2. D is substitutable for A ;
3. B is substitutable for E ; and
4. for all X in C , there exists Y in F such that X is substitutable for Y .

Arrow type coercions are invoked by wrapping the argument function of the coercion in a function expression that applies the appropriate coercions to the argument and result of the function. For example, assuming that f is a function from type A to type B the following coercion:

$(C \rightarrow D). \text{ coercion } (f)$

is rewritten to:

$\text{fn } x \Rightarrow D. \text{ coercion } (f(A. \text{ coercion } (x)))$

17.8 Automatic Widening

Syntax:

$\text{Coercion} ::= \text{widening coercion } \text{StaticParams? } (\text{Id IsType}) \text{CoercionClauses} = \text{Expr}$

Fortress supports what is sometimes called “widest-need” evaluation of numerical expressions. To see the problem, suppose that a and b are 32-bit floating-point numbers and c is a 64-bit floating-point number. It is easy, and tempting, to write an expression such as

$c = c + a \cdot b$

but if you stop to think about it, if interpreted naively it will compute the product $a \cdot b$ as a 32-bit floating-point number, and the impression that the sum may be accurate to the precision of a 64-bit floating-point number is only an illusion. Widest-need processing takes context into account and “widens” (or “upgrades”) the operands a and b to 64-bit floating-point numbers before the product is computed, so that the product will be computed as a 64-bit result.

Before widening is considered, a Fortress compiler, in its normal course of operation, analyzes an expression bottom-up. For every functional invocation, it needs to statically identify a specific declaration to be invoked. Once this is

done, then for every functional invocation, one of two conditions holds: (a) The type of the argument expression is a subtype of the type of the parameter in the identified declaration. (b) The type of the argument expression can be coerced to the type of the parameter in the identified declaration.

This process has to be done bottom-up because in order to select a specific declaration for a functional invocation, it is necessary to know the types of the argument expressions, and if an argument expression is itself a functional invocation, it is necessary to select the specific declaration for that invocation in order to find out the return type of the invocation.

Once an expression has been fully analyzed in this way, then widening is performed as a top-down process. For each functional invocation that appears in a context where a coercion can occur, ask whether the argument expression (which, remember, is a functional invocation) falls under case (a) or case (b) above. If case (b), coercion of the functional result is required; if the relevant coercion declaration is a “widening” coercion, then this functional invocation is a candidate for widening. Call the type of the functional invocation T , and call the expected type of the coercion context, U .

When the functional call was considered during the bottom-up analysis, in general many overloaded declarations may have been considered; of all those that were applicable with coercion to the static call, the most specific one was chosen. When considering widening, we consider the same set of declarations, but first discard all declarations whose return types are not subtypes of U . Then if any remain, and there is a unique most specific one among them, then that declaration is attached to the functional call instead. In this way a coercion step is eliminated (coercion is no longer needed to convert the result of the functional invocation from type T to type U), possibly at the expense of requiring a new coercion in a subexpression—but this is exactly the desired effect. Once this is done, then the subexpressions of the functional call are processed recursively.

This process is general enough not only to provide for widening floating-point precision, but to handle two other cases of interest as well: widening point computations to interval computations, and widening computations involving aggregate data structures whose components are widenable. For example, in $d(v \times w)$ suppose that d is of type $\mathbb{R}64$ and v and w are 3-vectors with components of type $\mathbb{R}32$; this strategy is capable of promoting v and w to 3-vectors with components of type $\mathbb{R}64$ and then performing all computations with $\mathbb{R}64$ precision.

Let’s go through the example of $c + a \cdot b$ in detail. The relevant declarations might look something like this:

```

trait  $\mathbb{R}32$ 
  ...
end
trait  $\mathbb{R}64$ 
  widening coercion ( $x:\mathbb{R}32$ ) = ...
  ...
end
opr  $\cdot$  ( $l:\mathbb{R}32, r:\mathbb{R}32$ ):  $\mathbb{R}32$  = ... (* 1 *)
opr  $\cdot$  ( $l:\mathbb{R}64, r:\mathbb{R}64$ ):  $\mathbb{R}64$  = ... (* 2 *)
opr  $+$  ( $l:\mathbb{R}32, r:\mathbb{R}32$ ):  $\mathbb{R}32$  = ... (* 3 *)
opr  $+$  ( $l:\mathbb{R}64, r:\mathbb{R}64$ ):  $\mathbb{R}64$  = ... (* 4 *)

```

The bottom-up analysis observes that a and b are both of type $\mathbb{R}32$, and discovers that declarations 1 and 2 are both applicable to the product, but declaration 1 would be chosen because it requires no coercion and declaration 2 would require coercion of both arguments to type $\mathbb{R}64$. The analysis then observes that c has type $\mathbb{R}64$ and the product expression has type $\mathbb{R}32$, so declaration 3 is not applicable but declaration 4 is applicable (though requiring a coercion from $\mathbb{R}32$ to $\mathbb{R}64$ for its second argument).

Now the top-down widening analysis is performed. The product is a functional call that is an argument to another functional call, and its result requires coercion, and that coercion is a widening coercion. Therefore the compiler reconsiders the applicable declarations, which were declarations 1 and 2. The result type of declaration 1 is not a subtype of $\mathbb{R}64$, but the result type of declaration 2 is a subtype of $\mathbb{R}64$ (in fact, it *is* $\mathbb{R}64$). Declaration 2 is

the most specific among the applicable declarations with that property (in fact, in this example it's the only one), so declaration 2 replaces declaration 1 for the product invocation. This in turn requires a and b to be coerced from type $\mathbb{R}32$ to $\mathbb{R}64$ before the product is performed.

Widening is a tricky business, best left to expert library designers. When used judiciously, it can greatly improve the precision of numerical computations without requiring a great deal of thought on the part of the application programmer and without cluttering up code with explicit conversions.

Future versions of this specification will include tables of coercions and widening coercions supported by the Fortress standard libraries.

Chapter 18

Dimensions and Units

Syntax:

<i>TypeRef</i> :	::=	<i>DimType</i>
<i>DimType</i>	::=	<i>DimRef</i>
		<i>TypeRef DimRef</i>
		<i>TypeRef</i> · <i>DimRef</i>
		<i>TypeRef</i> / <i>DimRef</i>
		<i>TypeRef</i> per <i>DimRef</i>
		<i>TypeRef UnitRef</i>
		<i>TypeRef</i> · <i>UnitRef</i>
		<i>TypeRef</i> / <i>UnitRef</i>
		<i>TypeRef</i> per <i>UnitRef</i>
		<i>TypeRef</i> in <i>DimRef</i>
<i>DimRef</i>	::=	<i>StaticArg</i>
<i>StaticArg</i>	::=	Unity
		dimensionless
		<i>StaticArg</i> · <i>StaticArg</i>
		<i>StaticArg StaticArg</i>
		<i>StaticArg</i> / <i>StaticArg</i>
		1 / <i>StaticArg</i>
		<i>StaticArg</i> ^ <i>StaticArg</i>
		<i>StaticArg</i> per <i>StaticArg</i>
		<i>DUPreOp StaticArg</i>
		<i>StaticArg DUPostOp</i>
		<i>TypeRef</i>
		(<i>StaticArg</i>)
<i>DUPreOp</i>	::=	square
		cubic
		inverse
<i>DUPostOp</i>	::=	squared
		cubed

```

Expr      ::= UnitExpr
UnitExpr  ::= UnitRef
           | Expr UnitRef
           | Expr · UnitRef
           | Expr / UnitRef
           | Expr per UnitRef
           | Expr in UnitRef
UnitRef   ::= StaticArg

```

There are special type-like constructs called *dimensions* that are separate from other types (described in Chapter 8). There are also special constructs that modify types and values called *units* that are instances of dimensions. These are used to describe physical quantities.

The Fortress standard libraries define dimensions and units for the standard SI system of measurement based on meters, kilograms, seconds, amperes, and so on (as described in Section 29.1). The Fortress standard libraries also provide supplemental units of measurement, such as feet and miles (as described in Section 29.2). For example, the Fortress standard libraries provide a dimension named `Length` whose default unit is named `meter` and abbreviated `m_`. By rendering convention, this abbreviation is rendered in roman type without the underscore: `m`. In contrast, the variable `m` is rendered as in standard mathematical notation: `m`. See Section 5.17 for a discussion of formatting conventions for tokens.

For readability, plural forms of the unit names are defined as equivalent to the corresponding singular forms; thus one can write `meters per second`, for example. Standard SI prefixes may be used on both the name and the symbol, so that `nanometer` and `nm` are also units of the dimension named `Length`, related to `meter` and `m` by a conversion factor of 10^{-9} .

Every dimension may have a default unit that is used for representing values of quantities of that dimension if no unit is specified explicitly. The Fortress standard libraries define these default dimensions and units:

<code>Length</code>	<code>meter</code>	<code>MagneticFlux</code>	<code>weber</code>
<code>Mass</code>	<code>kilogram</code>	<code>MagneticFluxDensity</code>	<code>tesla</code>
<code>Time</code>	<code>second</code>	<code>Inductance</code>	<code>henry</code>
<code>Frequency</code>	<code>hertz</code>	<code>Velocity</code>	<code>meters per second</code>
<code>Force</code>	<code>newton</code>	<code>Acceleration</code>	<code>meters per second squared</code>
<code>Pressure</code>	<code>pascal</code>	<code>Angle</code>	<code>radian</code>
<code>Energy</code>	<code>joule</code>	<code>SolidAngle</code>	<code>steradian</code>
<code>Power</code>	<code>watt</code>	<code>LuminousIntensity</code>	<code>candela</code>
<code>Temperature</code>	<code>kelvin</code>	<code>LuminousFlux</code>	<code>lumen</code>
<code>Area</code>	<code>square meter</code>	<code>Illuminance</code>	<code>lux</code>
<code>Volume</code>	<code>cubic meter</code>	<code>RadionuclideActivity</code>	<code>becquerel</code>
<code>ElectricCurrent</code>	<code>ampere</code>	<code>AbsorbedDose</code>	<code>gray</code>
<code>ElectricCharge</code>	<code>coulomb</code>	<code>DoseEquivalent</code>	<code>sievert</code>
<code>ElectricPotential</code>	<code>volt</code>	<code>AmountOfSubstance</code>	<code>mole</code>
<code>Capacitance</code>	<code>farad</code>	<code>CatalyticActivity</code>	<code>katal</code>
<code>Resistance</code>	<code>ohm</code>	<code>MassDensity</code>	<code>kilograms per cubic meter</code>
<code>Conductance</code>	<code>siemens</code>		

In addition, the Fortress standard libraries define the dimension `Information` with units `bit` and `byte` (and the plurals `bits` and `bytes`), a byte being equal to 8 bits. To avoid confusion, SI prefixes are *not* provided for these units; instead, programmers must use appropriate powers of 2 or 10, for example `106bits` or `220bits`.

Here are some examples of the use of dimensions and units:

```

x: ℝ64 Length = 1.3 m
t: ℝ64 Time = 5 s

```

```

v: ℝ64 Velocity = x/t
w: ℝ64 Velocity in nm/s = 17 nm/s
x: ℝ64 Velocity in furlongs per fortnight = v in furlongs per fortnight

```

Dimensions and units can be multiplied, divided, and raised to rational powers to produce new dimensions and units. Both the numerator and the denominator of a rational power of a dimension or a unit must be a valid `nat` *parameter* instantiation (as described in Section 11.2). Multiplication can be expressed using juxtaposition or `·`; division can be expressed using `/` or `per`. The syntactic operators `square` and `cubic` may be applied to a dimension or unit in order to raise it to the second power, third power, respectively; the special postfix syntactic operators `squared` and `cubed` may be used for the same purpose. The syntactic operator `inverse` may be applied to a dimension or unit to divide it into 1. All of these syntactic operators are merely syntactic sugar, expanded before type checking.

```

grams per cubic centimeter
meter per second squared
inverse ohms

```

One can also write $1/X$ as a synonym for X^{-1} if X is either a dimension or a unit.

Most numerical values in Fortress are dimensionless quantity values. Multiplying or dividing a dimensionless value by a unit produces a dimensioned value; thus `5 s` is the dimensioned value equal to five seconds, which has numerical value `5` and `second` for its unit. A dimensioned value may also be multiplied or divided by a unit, and the result is to combine the units; the expression `(17 nm)/s` first multiplies `17` by `nm` to produce the dimensioned value `seventeen nanometers`, which is then divided by the unit `s` to produce `seventeen nanometers per second`.

The unit of a dimensioned value may be changed to another unit of the same dimension by using the `in` operator, which takes a quantity to its left and a unit to its right. The `in` operator changes the unit and multiplies or divides the numerical value by an appropriate conversion constant so as to preserve the overall dimensioned value. Thus `1.3 m in nm` produces `1300000000 nm`.

Multiplying or dividing a dimensionless numerical type by a unit produces an equivalent dimensioned numerical type with that unit associated; thus `ℝ64 meter` is a type that is just like `ℝ64` but whose values are values of dimension `Length` measured in meters. A dimensioned numerical type may be further multiplied or divided by a unit. As a convenience, if a dimension has a default unit, a numerical type may also be multiplied or divided by a dimension, in which case the result is as if the default unit for that dimension had been used. The `in` operator may also be used to change the unit associated with a dimensioned type; in this situation the effect is merely to alter the unit associated with the type; no numerical operation is performed at run time.

Certain aggregate types, such as `Vector`, may also have associated units.

There are two reasons for using dimensions and units. One is that the `in` operator can supply conversion factors automatically. The other is that certain programming errors may be detected at compile time. When dimensioned values are added, subtracted, or compared, it is a static error if the units do not match. When dimensioned values are multiplied or divided, their units are multiplied or divided. When taking the square root of a dimensioned value, the unit of the result is the square root of the argument's unit. Other numerical functions, such as `sin` and `log`, require dimensionless arguments.

A variable whose declared type includes a dimension without an accompanying unit is understood to have the default unit for that dimension. Thus, in most cases, the runtime unit of an expression can be statically inferred. However, there are exceptions. For example, consider the following declaration:

```
a : Object[3] = [5 mg, 3 m, 4 s]
```

Now suppose we declare a function that takes an array of objects and returns one of its elements:

```
f(xs : Object[3]) = xs1
```

The value of the call $f(a)$ is 3 m . However, the static type of $f(a)$ is simply Object . When the value 3 m is placed in an array of objects, the value is boxed, and the unit associated with the value must be included as part of the boxed value. However, unboxed values need not include unit information at runtime, as this information is statically evident in such cases.

There are also special static parameters for units and dimensions; see Section 11.4.

Chapter 19

Tests and Properties

Fortress supports automated program testing. Components and APIs can declare *tests*. Test declarations specify explicit finite collections over which the test is run. Components, APIs, traits, and objects can declare *properties* which describe boolean conditions that the enclosing construct is expected to obey. Tests and properties may modify the program state.

19.1 The Purpose of Tests and Properties

To help make programs more robust, Fortress programs are allowed to include special constructs called *tests* and *properties*. Tests consist of test data along with code that can be run on that data. Properties are documentation used to describe the behavior of the traits and functions of a program; they can be thought of as comments written in a formal language. For each property, there is a special function that can be called by a program's tests to ensure that the property holds for specific test data.

A fortress includes hooks to allow programmers to run a specific test on an executable component, and to run all tests on such a component. A particularly useful time to run the tests of an executable component is at component link time; errors in the behavior of constituent components can be caught before the linked program is run.

19.2 Test Declarations

Syntax:

```
TestDecl ::= test Id [GeneratorList] = Expr
GeneratorList ::= Generator( , Generator)*
Generator ::= Id ← Expr
              | ( Id , IdList ) ← Expr
              | Expr
IdList ::= Id( , Id)*
```

A test declaration begins with `test` followed by an identifier, a list of zero or more generators (described in Section 13.14) enclosed in square brackets, the token `=`, and a subexpression. When a test is *run* (See Section 22.7), the subexpression is evaluated in each extension of the enclosing environment corresponding to each point in the cross product of bindings determined by the generators in the generator list.

For example, the following test:

```
test fxLessThanFy[x ← E, y ← F] = assert(f(x) < f(y))
```

checks that, for each value x supplied by generator E , and for each value y supplied by generator F , $f(x)$ is less than $f(y)$.

19.3 Other Test Constructs

Syntax:

```
UniversalMod ::= test
```

The `test` modifier can also appear on a function definition, trait definition, object definition, or top-level variable definition. In these contexts, the modifier indicates that the program construct it modifies can be referred to by a test. Functions with modifier `test` must not be overloaded with functions that do not have modifier `test`, and traits with modifier `test` must not be extended by traits or objects that do not have modifier `test`. The collection of all constructs in a program that include modifier `test` are referred to collectively as the program's *tests*. Tests can refer to non-tests but it is a static error for a non-test to refer to any test.

For example, we can write the following helper function:

```
test ensureApplicationFails(g, x) = do
  applicationSucceeded := false
  try
    g(x)
    applicationSucceeded := true
  catch e
    Exception ⇒ ()
  end
  if applicationSucceeded then
    fail "Application succeeded!"
  end
end
```

The library function `fail` displays the error message provided and terminates execution of the enclosing test.

19.4 Running Tests

When a program's tests are *run*, the following actions are taken:

1. All top-level definitions, including constructs beginning with modifier `test`, are initialized. A test declaration with name t declares a function named t that takes a tuple of parameters corresponding to the variables bound in the generator list of the test declaration. For each variable v in the generator list of t , if the type of generator supplied for v is `Generator[α]` then the parameter in the function corresponding to v has type α . The return type of the function is `()`.

Each such function bound in this manner is referred to as a *test function*. Test functions can be called from the rest of the program's tests.

2. Each test t in a program is run in each extension of t 's enclosing environment with a point in the cross product of bindings determined by the generators in the test's generator list.

19.5 Test Suites

In order to allow programmers to run strict subsets of all tests defined in a program, Fortress allows tests to be assembled into *test suites*.

The convenience object `TestSuite` is defined in the Fortress standard libraries. An instance of this object contains a set of test functions that can all be called by invoking the method `run`:

```
test object TestSuite(testFunctions = {})
  add(f: () → ()) = testFunctions.insert(f)
  run() =
    for t ← testFunctions do
      t()
    end
end
```

Note that all tests in a `TestSuite` are run in parallel.

19.6 Property Declarations

Syntax:

```
PropertyDecl ::= property (Id =)? (∀ ValParam)? Expr
ValParam     ::= BindId
              | (Params?)
BindId       ::= Id
              | -
Params       ::= (Param ,)* (Varargs ,)? Keyword( , Keyword)*
              | (Param ,)* Varargs
              | Param( , Param)*
VarargsParam ::= Id : TypeRef ...
Varargs      ::= VarargsParam
Keyword      ::= Param = Expr
PlainParam   ::= BindId IsType?
              | TypeRef
Param        ::= PlainParam
```

Components and APIs may include *property* declarations, documenting boolean conditions that a program is expected to obey. Syntactically, a property declaration begins with `property` followed by an optional identifier followed by the token `=`, an optional value parameter, which may be a tuple, preceded by the token `∀`, and a boolean subexpression. In any execution of the program, the boolean subexpression is expected to evaluate to true at any time for any binding of the property declaration's parameter to any value of its declared type. When a property declaration includes an identifier, the property identifier is bound to a function whose parameter and body are that of the property, and whose return type is `Boolean`. A function bound in this manner is referred to as a *property function*.

Properties can also be declared in trait or object declarations. Such properties are expected to hold for all instances of the trait or object and for all bindings of the property's parameter. If a property in a trait or object includes a name, the name is bound to a method whose parameter and body are that of the property, and whose return type is `Boolean`. A method bound in this manner is referred to as a *property method*. A property method of a trait *T* can be called, via dotted method notation, on an instance of *T*.

Property functions and methods can be referred to in a program's tests. If the result of a call to a property function or method is not *true*, a `TestFailure` exception is thrown. For example, we can write:

```
property flsMonotonic =  $\forall(x:\mathbb{Z}, y:\mathbb{Z}) (x < y) \rightarrow (f(x) < f(y))$   
  
test s : Set[ $\mathbb{Z}$ ] = {-2000, 0, 1, 7, 42, 59, 100, 1000, 5697}  
test flsMonotonicOverS[x ← s, y ← s] = flsMonotonic(x, y)  
test flsMonotonicHairy[x ← s, y ← s] = flsMonotonic(x, x2 + y)
```

The test `flsMonotonicOverS` tests that function f is monotonic over all values in set s . The test `flsMonotonicHairy` tests that f is monotonic with respect to the values in s compared to the set of values corresponding to all the ways in which we can choose an element of s , square it, and add it to another element of s .

Chapter 20

Type Inference

Type inference in Fortress is performed independently on each simple component (described in Chapter 22). For separation of concerns, in this chapter, we describe the Fortress type inference as a procedure performed over a whole Fortress program. We explain how this procedure can be adapted to perform type inference over a simple program component in Section 22.5. Note that type inference cannot be performed on each functional (function or method) declaration in isolation because it may be declared mutually recursively or may contain free variables.

20.1 What Is Inferred

Types of functional parameters, functional results, and variables may be elided in a program where they can be inferred automatically. Similarly, instantiations of static parameters of generic functional invocations may be elided where they can be inferred automatically. A Fortress compiler must allow types and static arguments inferable via the procedure described in Section 20.2 to be elided, no more and no less. This strict requirement is made for the sake of source-code portability; it is important that a program that compiles on one compiler will compile on all compilers. Of course, there is nothing preventing a development environment from aiding programmers by performing more sophisticated analyses and filling in additional information, but the text produced is not a valid Fortress program unless all elided types and static arguments can be inferred via the described procedure.

20.2 Type Inference Procedure

To perform type inference, we first infer any elided parameter type in each functional declaration that can be inferred from other declaration as follows:

1. If the declaration is a functional declaration and the type of the declared functional is declared via a separate declaration, any elided parameter type is inferred to have the type provided by the separate declaration.
2. Otherwise, if the declaration is a method declaration that overloads a method declaration provided by a super-trait, any elided parameter type is inferred to have the type provided (or inferred) by the overloaded method declaration.

All remaining parameter types are inferred along with instantiations of static parameters.

In the following, we adopt the convention of writing “primed” static variables, $T'_0, \dots, T'_m, U'_0, \dots$, to stand for fresh static variables. Our procedure will introduce primed static variables as placeholders that are to be replaced with

nonprimed types and static arguments before the termination of type inference. We abuse notation by using *types* to refer to both types and static arguments when it is clear from context.

First, we annotate every expression e that is not a functional application with a fresh static variable (written here as a superscript):

$$e^{T'}$$

and every functional application $f(a_0, \dots, a_n)$ (where f is the name of a generic functional) with fresh static variables for the instantiation of each of the functional's static parameters as well as a superscript:

$$f[[T'_0, \dots, T'_{m-1}]](a_0, \dots, a_n)^{T'_m}$$

and every functional parameter x without a declared type with fresh static variable as its declared type:

$$x : T'$$

We say that a functional application is an *outermost* functional application if and only if it is not a proper subexpression of another functional application.

For each outermost functional application:

$$f[[T'_0, \dots, T'_{m-1}]](a_0, \dots, a_n)^{T'_m}$$

let $f[[R_0, \dots, R_{m-1}]](p_0 : S_0, \dots, p_n : S_n) : R_m$ be a declaration of f . Some of the types and static parameters appearing at the declaration might be primed static variables themselves.

Note that there may be several declarations of f due to overloading. When there are multiple declarations for f , type inference is performed to each declaration. Only the declarations to which type inference succeed are considered for overload resolution.

We accumulate a table of subtyping constraints as follows:

First, we add the following constraint to our table:

$$T'_m <: [R_0 \mapsto T'_0, \dots, R_{m-1} \mapsto T'_{m-1}]R_m$$

where we use the notation $[R_0 \mapsto T'_0, \dots, R_{m-1} \mapsto T'_{m-1}]$ to denote the safe substitution of R_0, \dots, R_{m-1} with T'_0, \dots, T'_{m-1} .

If the functional application occurs as the right-hand side of a declaration $x : T = e$, we add the constraint:

$$T'_m <: T$$

An analogous constraint is added if the application occurs as the right-hand-side of an assignment to a variable with type T , or as the body of a functional with return type T , or as an expression ascripted with type T , etc. We refer to all of these cases collectively by saying that the functional application “occurs in the context of type T ”.

Additionally, for every argument $a_i^{U'_i}$ corresponding to value parameter p_i , we add the following constraint:

$$U'_i <: [R_0 \mapsto T'_0, \dots, R_{m-1} \mapsto T'_{m-1}]S_i$$

Additionally, we add all “nested constraints” accumulated for a_i , where nested constraints are accumulated as follows:

1. If a_i is not a functional application or a `typecase` expression, the nested constraints are the union of the nested constraints of all subexpressions of a_i plus the constraint $R <: U'_i$, where R is the type of a_i in the enclosing context.
2. If a_i is a `typecase` expression, only nested constraints common to all branches are accumulated. Furthermore, static parameter instantiations of generic functional calls within each branch must be inferred independently, in a separate table extending the constraints of the enclosing context with only those constraints available in that branch.

3. If a_i is a functional application, the nested constraints are all constraints accumulated for a_i as if it were an outermost generic functional application occurring in context U'_i .

Once we have accumulated all constraints for each outermost functional application, the constraints are solved in the following phases:

1. For every cycle of constraints of naked primed static variables $T'_1 <: \dots <: T'_n <: T'_1$, one of T'_1, \dots, T'_n is chosen. We call the chosen naked primed static variable T^* . All occurrences of T'_1, \dots, T'_n in the constraints and throughout the program are replaced with T^* . Redundant constraints are eliminated. This process is repeated until a fixed point is reached.
2. For every naked primed static variable T' , consider the set of all subtype constraints on T' : $T' <: U_1, \dots, T' <: U_n$. The *unexpanded* inferred least upper bound for T' is the intersection type (described in Section 8.8) $U_1 \cap \dots \cap U_n$.
3. If some primed static variable R' appears in one of the U_i in the unexpanded inferred least upper bound for T' and R' does not appear within a type constructor, R' is replaced with its own unexpanded inferred type. This process is repeated until a fixed point is reached.

If T' is expanded to a type with no primed variables, this phase is done. If T' is expanded to a type containing no primed variables except T' , the expansion E of T' is replaced with a *fixed-point type* $\mu T'.E$. Fixed-point types are needed to express types that would be “infinitely” large if expanded out into conventional ground types. We have the following property for fixed-point types: $\mu X.E = [X \mapsto \mu X.E]E$. Otherwise, T' is inferred to have type Object.

4. For every naked primed static variable T' , consider the set of all supertype constraints on T' : $V_1 <: T', \dots, V_n <: T'$. The unexpanded inferred greatest lower bound for T' is the union type (described in Section 8.8) $V_1 \cup \dots \cup V_n$. This type is expanded analogously to the procedure used for the intersection of T' .
5. The inferred type for a static variable T' is the intersection of the expanded inferred least upper bound and the expanded inferred greatest lower bound for T' . For the purposes of type checking, the inferred type is put into clausal normal form as a canonical form, eliminating redundant clauses.

20.3 Finding “Closest Expressible Types” for Inferred Types

Once solutions to the primed variables are inferred, we must find the *closest expressible type* to each inferred type. Expressible types are all types that are neither intersection types nor union types and that syntactically contain only expressible types. Note that fixed-point types are expressible types as in the Java type inference.

1. Given an intersection type $S_1 \cap \dots \cap S_n$, if there is a unique most general expressible type T such that T is a subtype of S_1, \dots, S_n , then T is the closest expressible type to $S_1 \cap \dots \cap S_n$. If there is not a unique most general expressible type, then the close expressible type is the closest expressible supertype of the multiple closest expressible types; this process is guaranteed to terminate because the type hierarchy is rooted at type Object.
2. Given a union type $S_1 \cup \dots \cup S_n$, if there is a unique most specific expressible type T such that T is a supertype of S_1, \dots, S_n , then T is the closest expressible type to $S_1 \cup \dots \cup S_n$. If there is not a unique most general expressible type, then the close expressible type is the closest expressible supertype of the multiple closest expressible types.
3. Given a generic type instantiation $C[[S_1, \dots, S_n]]$, where some of the S_1, \dots, S_n are not expressible, the closest expressible type is $C[[E(S_1), \dots, E(S_n)]]$ where $E(S_i)$ is the closest expressible type to S_i .

Each primed type in a program is replaced with the closest expressible type to its solution. Any remaining primed types that have not yet been inferred are then inferred to have type Object.

Chapter 21

Memory Model

Fortress programs are highly multithreaded by design; the language makes it easy to expose parallelism. However, many Fortress objects are mutable; without judicious use of synchronization constructs—reductions and atomic expressions—*data races* will occur and programs will behave in an unpredictable way. The memory model has two important functions:

1. Define a programming discipline for the use of data and synchronization, and describe the behavior of programs that obey this discipline. This is the purpose of Section 21.2.
2. Define the behavior of programs that do not obey the programming discipline. This constrains the optimizations that can be performed on Fortress programs. The remaining sections of this chapter specify the detailed memory model that Fortress programs must obey.

21.1 Principles

The Fortress memory model has been written with several important guiding principles in mind. Violations of these principles should be taken as a flaw in the memory model specification rather than an opportunity to be exploited by the programmer or implementor. The most important principle is this: violations of the Fortress memory model must still respect the underlying data abstractions of the Fortress programming language. All data structures must be properly initialized before they can be read by another thread, and a program must not read values that were never written. When a program fails, it must fail gracefully by throwing an exception.

The second goal is nearly as important, and much more difficult: present a memory model which can be understood thoroughly by programmers and implementors. It should never be difficult to judge whether a particular program behavior is permitted by the model. Where possible, it should be possible to check that a program obeys the programming discipline.

The final goal of the Fortress memory model is to permit aggressive optimization of Fortress programs. A multi-processor memory model can rule out optimizations that might be performed by a compiler for a uniprocessor. The Fortress memory model attempts to rule out as few behaviors as possible, but more importantly attempts to make it easy to judge whether a particular optimization is permitted or not. The semantics of Fortress already allows permissive operation reordering in many programs, simply by virtue of the implicitly parallel semantics of tuple evaluation and looping.

21.2 Programming Discipline

If Fortress programmers obey the following discipline, they can expect sequentially consistent behavior from their Fortress programs:

- Updates to shared locations should always be performed using an `atomic` expression. A location is considered to be shared if and only if that location can be accessed by more than one thread at a time. For example, statically partitioning an array among many threads need not make the array elements shared; only elements actually accessed by more than one thread are considered to be shared.
- Within a thread or group of implicit threads objects should not be accessed through aliased references; this can yield unexpected results. Section 21.2.3 defines the notion of *apparently disjoint* references. An object should not be written through one reference when it is accessed through another apparently disjoint reference.

The following stylistic guidelines reduce the possibility of pathological behavior when a program strays from the above discipline:

- Where feasible, reduction should be used in favor of updating a single shared object.
- Immutable fields and variables should be used wherever practical. We discuss this further in Section 21.2.1.
- A getter or a setter should behave as though it is performing a simple field access, even if it internally accesses many locations. The simplest (but not necessarily most efficient) way to obtain the appropriate behavior is to make hand-coded accessors `atomic`. Section 21.2.2 expands on this.

21.2.1 Immutability

Recall from Section 4.3 that we can distinguish mutable and immutable memory locations. Any thread that reads an immutable field will obtain the initial value written when the object was constructed. In this regard it is worth re-emphasizing the distinction between an object reference and the object it refers to. A location that does not contain a value object contains an object reference. If the field is immutable, that means the reference is not subject to change; however, the object referred to may still be modified in accordance with the memory model.

By contrast, recall that all the fields of a value object are immutable; however, a mutable location may have a value type. Such a location can be written, completely replacing the value object it contains. Similarly, reading the value contained in such a location conceptually causes the entire value object to be read; if this isn't followed by an immediate field reference, the read must be performed atomically. This may potentially be expensive (see Section 21.3).

21.2.2 Providing the Appearance of a Single Object

In practice, most accesses to fields occur through a mediating getter or setter method. It should not be possible for the programmer to tell whether a given getter or setter directly accesses a field or if it performs additional computations. Similarly, any subscripting operation must be indistinguishable from accessing a single field. Thus, accessor methods and subscripting methods should provide the appearance of atomicity, as described in Section 21.3. The Fortress standard libraries are written to preserve this abstraction. For example, the `Array` type in Fortress makes use of one or more underlying `HeapSequences`, and array subscripting preserves the atomicity guarantees of this underlying object.

21.2.3 Modifying Aliased Objects

In common with Fortran, and unlike most other popular programming languages, Fortress gives special treatment to accesses to a location through different aliases. For the purposes of variable aliasing, it is important to define the notion

of *apparently disjoint* (or simply disjoint) object and field references. If two references are not disjoint, we say they are *certainly the same*, or just *the same*. By contrast, we say object references are *identical* if they refer to the same object, and *distinct* otherwise. Accesses to fields reached via apparently disjoint object references may be reordered (except an initializing write is never reordered with respect to other accesses to the identical location).

Distinct references are always disjoint. Two identical references are apparently disjoint if they are obtained from any of the following locations:

- distinct parameters of a single function call
- distinct fields
- a parameter and a field
- identically named fields read from apparently disjoint object references
- distinct reads of a single location for which there may be an interposing write

When comparing variables defined in different scopes, these rules will eventually lead to reads of fields or to reads of parameters in some common containing scope.

We extend this to field references as follows: two field references are apparently disjoint if they refer to distinct fields, or they refer to identically named fields read from apparently disjoint object references.

Consider the following example:

```
f(x : ℤ64[2], y : ℤ64[2]) : ℤ64 = do
  x0 := 17
  y0 := 32
end
```

Here x and y in f are apparently disjoint; the writes may be reordered, so the call $f(a, a)$ may assign either 17 or 32 to a_0 .

A similar phenomenon occurs in the following example:

```
g(x : ℤ64[2], y : ℤ64[2]) : ℤ64 = do
  x0 := 17
  y0
end
```

Again x and y are apparently distinct in g , so the write to x_0 and the read of y_0 may be reordered. The call $g(a, a)$ will assign 17 to a_0 but may return either the former value of a_0 or 17.

It is safe to *read* an object through apparently disjoint references:

```
h(x : ℤ64[2], y : ℤ64[2]) : ℤ64 = do
  u : ℤ64 = x0
  v : ℤ64 = y0
  u + v
end
```

A call to $h(a, a)$ will read a_0 twice without ambiguity. Note, however, that the reads may still be reordered, and if a_0 is written in parallel by another thread this reordering can be observed.

If necessary, `atomic` expressions can be used to order disjoint field references:

```
f'(x : ℤ64[2], y : ℤ64[2]) : () = do
  atomic x0 := 17
  atomic y0 := 32
end
```

Here the call $f(a, a)$ ends up setting a_0 to 32. Note that simply using a single `atomic` expression containing one or both writes is not sufficient; the two writes must be in distinct `atomic` expressions to be required to occur in order.

When references occur in distinct calling contexts, they are disambiguated at the point of call:

```

j(x : ℤ64[2], y : ℤ64) : () = x_0 := y
k(x : ℤ64[2]) : () = do
  j(x, 17)
  j(x, 32)
end
l(x : ℤ64[2], y : ℤ64[2]) : () = do
  j(x, 17)
  j(y, 32)
end

```

Here if we call $k(a)$ the order of the writes performed by the two calls to j is unambiguous, and a_0 is 32 in the end. By contrast, $l(a, a)$ calls j with two apparently disjoint references, and the writes in these two calls may thus be reordered.

21.3 Read and Write Atomicity

Any read or write to a location is *indivisible*. In practical terms, this means that each read operation will see exactly the data written by a single write operation. Note in particular that indivisibility holds for a mutable location containing a large value object. It is convenient to imagine that every access to a mutable location is surrounded by an `atomic` expression. However, there are a number of ordering guarantees provided by `atomic` accesses that are not respected by non-`atomic` accesses.

21.4 Ordering Dependencies among Operations

The Fortress memory model is specified in terms of two orderings: dynamic program order (see Chapter 4 and Chapter 13) and memory order. The actual order of memory operations in a given program execution is *memory order*, a total order on all memory operations. Dynamic program order constrains memory order. However, memory operations need not be ordered according to dynamic program order; many memory operations, even reads and writes to a single field or array element, can be reordered. Programmers who adhere to the model in Section 21.2 can expect sequentially consistent behavior: there will be a global ordering for all memory operations that respects dynamic program order.

Here is a summary of the salient aspects of memory order:

- There is a single memory order which is respected in all threads.
- Every read obtains the value of the immediately preceding write to the identical location in memory order.
- Memory order on `atomic` expressions respects dynamic program order.
- Memory order respects dynamic program order for operations that certainly access the same location.
- Initializing writes are ordered before any other memory access to the same location.

21.4.1 Dynamic Program Order

Much of the definition of *dynamic program order* is given in the descriptions of individual expressions in Chapter 13. It is important to understand that dynamic program order represents a conceptual, naive view of the order of operations

in an execution; this naive view is used to define the more permissive memory order permitted by the memory model. Dynamic program order is a partial order, rather than a total order; in most cases operations in different threads will not be ordered with respect to one another. There is an important exception: there is an ordering dependency among threads when a thread starts or must be complete.

An expression is ordered in dynamic program order after any expression it dynamically contains, with one exception: a `spawn` expression is dynamically ordered before any subexpression of its body. The body of the `spawn` is dynamically ordered before any point at which the spawned thread object is observed to have completed.

Only expressions whose evaluation completes normally occur in dynamic program order, unless the expression is “directly responsible” for generating abrupt termination. Examples of the latter case are `throw` and `exit` expressions and division by zero. In particular, when the evaluation of a subexpression of an expression completes abruptly, causing the expression itself to complete abruptly, the containing expression does not occur in dynamic program order. A `label` block is ordered after an `exit` that targets it. The expressions in a `catch` clause whose `try` block throws a matching exception are ordered after the `throw` and before any expression in the `finally` clause. If the `catch` completes normally, the `try` block as a whole is ordered after the expressions in the `finally` clause. For this reason, when we refer to the place of non-`spawn` expression in dynamic program order, we mean the expression or any expression it dynamically contains.

For any construct giving rise to implicit threads—tuple evaluation, function or method call, or the body of an expression with generators such as `for`—there is no ordering in dynamic program order between the expression executed in each thread in the group. These subexpressions are ordered with respect to expressions which precede or succeed the group.

When a function or method is called, the body of the function or method occurs dynamically after the arguments and function or receiver; the call expression is ordered after the body of the called function or method.

For conditional expressions such as `if`, `case`, and `typecase`, the expression being tested is ordered dynamically before any chosen branch. This branch is in turn ordered dynamically before the conditional expression itself.

Iterations of the body of a `while` loop are ordered by dynamic program order. Each evaluation of the guarding predicate is ordered after any previous iteration and before any succeeding iteration. The `while` loop as a whole is ordered after the final evaluation of the guarding predicate, which yields *false*.

An iteration of the body of a `for` loop, and each evaluation of the body expression in a comprehension or big operator, is ordered after the generator expressions.

21.4.2 Memory Order

Memory order gives a total order on all memory accesses in a program execution. A read obtains the value of the most recent prior write to the identical location in memory order. In this section we describe the constraints on memory order, guided by dynamic program order. We can think of these constraints as specifying a partial order which must be respected by memory order. The simplest constraint is that accesses certainly to the same location must respect dynamic program order. Apparently disjoint accesses need not respect dynamic program order, but an initializing write must be ordered before all other accesses to the identical location in program order.

Accesses in distinct (non-nested) `atomic` expressions respect dynamic program order. Given an `atomic` expression, we divide accesses into four classes:

1. Components, dynamically contained within the `atomic` expression.
2. Ancestors, dynamically ordered before the `atomic` expression.
3. Descendants, dynamically ordered before the `atomic` expression.
4. Peers, dynamically unordered with respect to operations dynamically contained within the `atomic` expression.

We say an `atomic` expression is *effective* if it contains an access to a location, there is a peer access to the identical location, and at least one of these accesses is a write. For an effective `atomic` expression, every peer access must either be a *predecessor* or a *successor*. A predecessor must occur before every component and every descendant in memory order. A successor must occur after every component and every ancestor in memory order. Every ancestor must occur before every descendant in memory order.

The above conditions guarantee that there is a single, global ordering for the effective `atomic` expressions in a Fortress program. This means that for any pair of `atomic` expressions A and B one of the following conditions holds:

- A is dynamically contained inside B .
- B is dynamically contained inside A .
- Every expression dynamically contained in A precedes every expression dynamically contained in B in memory order. This will always hold when A is dynamically ordered before B .
- Every expression dynamically contained in B precedes every expression dynamically contained in A in memory order. This will always hold when B is dynamically ordered before A .

The above rules are also sufficient to guarantee that `atomic` expressions nested inside an enclosing `atomic` behave with respect to one another just as if they had occurred at the top level in an un-nested context.

Any access preceding a `spawn` in dynamic program order will precede accesses in the spawned expression in memory order. Any access occurring after a spawned thread has been observed to complete in dynamic program order will occur after accesses in the spawned expression in memory order.

A reduction variable in a `for` loop does not have a single associated location; instead, there is a distinct location for each loop iteration, initialized by writing the identity of the reduction. These locations are distinct from the location associated with the reduction variable in the surrounding scope. In memory order there is a read of each of these locations each of which succeeds the last access to that variable in the loop iteration, along with a read of the location in the enclosing scope which succeeds all accesses to that location preceding the loop in dynamic program order. These reads are followed by a write of the location in the enclosing scope which in turn precedes all accesses to that location that succeed the loop in dynamic program order.

Finally, reads and writes in Fortress programs must respect dynamic program order for operations that are *semantically related*. If the read A precedes the write B in dynamic program order, and the value of B can be determined in some fashion without recourse to A , then these operations are not semantically related. A simple example is if A is a reference to variable x and B is the assignment $y := x \cdot 0$. Here it can be determined that $y := 0$ without recourse to x and these variables are not semantically related. By contrast, the write $y := x$ is always semantically related to the read of x . Note that two operations can only be semantically related if a transitive data or control dependency exists between them.

Chapter 22

Components and APIs

Fortress programs are developed, compiled, and deployed as *encapsulated upgradable components* that exist not only as programming language features, but also as self-contained run-time entities that are managed throughout the life of the software. The imported and exported references of a component are described with explicit *APIs*. With components and APIs, Fortress provides the stability benefits of static linking with the sharing and upgrading benefits of dynamic linking.¹ In addition to an informal description of the component system in this chapter, we also formally specify key functionality of the system, and illustrate how we can reason about the correctness of the system in Appendix C.

22.1 Overview

Syntax:

```
File ::= CompilationUnit
      | Imports? Exports Decls?
      | Imports? AbsDecls
      | Imports AbsDecls?
CompilationUnit ::= Component
                 | Api
```

Components are the fundamental structure of Fortress programs. They export and import APIs, which serve as “interfaces” of the components. Components do not refer directly to other components. Rather, all external references are to APIs imported by the component. These references are resolved by linking components together: the references of a component to an imported API are resolved to a component that exports that API. Linking components produces new components, whose *constituents* are the components that were linked together.

Components are similar to modules in other programming languages, such as those of ML and Scheme [20, 16, 15]. But, unlike modules in those languages, components are designed for use during both development and deployment of software. In addition to compilation and linking, components can be produced by upgrading one component using another component that exports some of the APIs exported by the first component.

A key aspect of Fortress components is that they are encapsulated, so that upgrading one component does not affect any other component, even those produced by linking with the component that was upgraded. Abstractly, each component has its own copy of its constituents. However, implementations are expected to share common constituents when possible.

Users do not manipulate components directly. Instead, every component is installed in a persistent database on the system. We think of this database, which we call a *fortress*, as the agent that actually performs operations such as

¹The system described in this chapter is based on that described in [2].

compilation, linking, upgrading, and execution of components: a virtual machine, a compiler, and a library registry all rolled into one. A fortress also maintains a list of APIs that are installed on it. A fortress also provides a shell by which the user can issue commands to it. Components and APIs are immutable objects. A fortress maps names to components installed on the system. The fortress operations are modeled as methods of the fortress that change the mapping.

The ways in which fortresses are actually realized on particular platforms are beyond the scope of this specification. An implementor might choose to instantiate a fortress as a process, or as a persistent object database stored in a file system, with fortress operations being implemented as scripts that manipulate this database.

We call the source code for a single software component a *project*. Typically, when a project written in other programming languages is compiled, each file in the project is separately compiled. To ship an application, these files are linked together to form an application or library. Fortress uses a different model: a project is compiled directly into a single component, which is installed in the compiling fortress.

From the point of view of the compiler, all the source code for a project is contained in a single file. This approach simplifies the design but it is unworkable for components of substantial size. Therefore, the compiler can be instructed to concatenate several source files together before compiling, while maintaining the original source location information.

After these components are compiled from source files, they can then be linked together to form larger components.

22.2 Components

Syntax:

```

Component ::= component DottedId Imports? Exports Decls? end
Imports ::= Import+
Import ::= import ImportFrom
           | import AliasedDottedIds
ImportFrom ::= * (except Names)? from DottedId
           | AliasedNames from DottedId
Names ::= Name
           | { NameList }
NameList ::= Name( , Name)*
AliasedNames ::= AliasedName
           | { AliasedNameList }
AliasedName ::= Id ( as DottedId)?
           | opr Op ( as Op)?
           | opr LeftEncloser RightEncloser ( as LeftEncloser RightEncloser)?
AliasedNameList ::= AliasedName( , AliasedName)*
AliasedDottedIds ::= AliasedDottedId
           | { AliasedDottedIdList }
AliasedDottedId ::= DottedId ( as DottedId)?
AliasedDottedIdList ::= AliasedDottedId( , AliasedDottedId)*
Exports ::= Export+
Export ::= export DottedIds
DottedIds ::= DottedId
           | { DottedIdList }
DottedIdList ::= DottedId( , DottedId)*
Decls ::= Decl+

```

In this specification, we will refer to components created by compiling a file as *simple components*, while components created by linking components together will be known as *compound components*.

The source code of a simple component definition begins with `component` followed by a *possibly qualified name* (an identifier or a sequence of identifiers separated by periods with no intervening whitespace), followed by a sequence of *import* statements, and a sequence of *export* statements, and finally a sequence of declarations.

An import statement either imports an API and allows the specified names (separated by commas) declared in the API to be referred to with their unqualified names:

```
import {name+} from apiName
```

or imports an API as another name:

```
import apiName as anotherAPIName
```

For convenience, an import statement can import an API and allow all elements declared in that API to be referred to with unqualified names:

```
import * from apiName
```

or can import an API and allow all elements except the specified names (separated by commas) declared in that API to be referred to with unqualified names:

```
import * except {name+} from apiName
```

If multiple elements with conflicting names are imported from separate APIs, all references to those elements within the component definition must be fully qualified. An export statement specifies the APIs that the component exports.

Every component implicitly imports the Fortress core APIs; every fortress has at least one component implementing all of these APIs. A *preferred* component exporting these APIs (configurable by the user) is implicitly linked to every component installed in the fortress.

An API (described in Section 22.3) is an interface of a component. A component must provide a declaration, or a set of declarations, that *satisfies* every top-level declaration in any API that it exports, as described below. A component may include declarations that do not participate in satisfying any exported declaration (i.e., a declaration of any exported API), and in particular, no declaration modified by `private` participates in satisfying any exported declaration. Such declarations are not visible from outside the component.

Type aliases and dimension, unit, test and property declarations are satisfied only by declarations that are the same.²

A top-level variable declaration declaring a single variable is satisfied by any top-level variable declaration that declares the name with the same type (in the component, the type may be inferred). A top-level variable declaration declaring multiple variables is satisfied by a set of declarations (possibly just one) that declare all the names with their respective types (which again, may be inferred). In either case, the mutability of a variable must be the same in the exported and satisfying declarations. A top-level variable declaration declaring a single immutable variable with arrow type $P \rightarrow R$ is also satisfied by a set of top-level function and functional method declarations if the union of their parameter types is P and each of their return types is a subtype of R . Thus, an abstract function declaration in the form of variable declarations in an API need not be replicated in a component exporting that API.

A top-level function declaration is satisfied by a set of top-level function and functional method declarations the union of whose parameter types is the parameter type of the exported declaration, and each of whose return types is a subtype of the return type of the exported declaration.

A trait or object declaration is satisfied by a declaration that has the same header,³ and contains, for each field declaration and non-abstract method declaration in the exported declaration, a satisfying declaration (or a set of declarations). If a trait declaration (in an API) has a `comprises` clause containing "...", then the `comprises` clause of its satisfying declaration must contain all the types listed in the declaration in the API but it cannot contain "..."; instead, it

²The declarations are compared after ASCII conversion and scanning, and differences in the characters within whitespace elements is permitted provided they do not change whether the whitespace element is line-breaking.

³The order of the modifiers, the clauses, and the types in the `extends`, `excludes` and `comprises` clauses may differ.

may contain other types declared in the component provided that those types are not declared by the API. A field declaration is satisfied by a (possibly implicit) getter and/or setter declaration, depending on whether the field is declared to be `hidden` and/or `settable` by the exported declaration. A method declaration is satisfied by a set of method declarations the union of whose parameter types is the parameter type of the exported declaration, and each of whose return types is a subtype of the return type of the exported declaration. A satisfying trait or object declaration may contain method and field declarations not exported by the API but these may not be overloaded with method or field declarations provided by (contained in or inherited by) any declarations exported by the API.

For functional declarations, recall that several functional declarations may define the same entity (i.e., they may be overloaded). Given a set of overloaded declarations, it is not permitted to export some of them and not others.

When a component is compiled, the APIs it refers to must be present in the fortress. The import statements in a component are not a way to abbreviate unqualified names of objects or functions. In our system, an import statement merely allows references to the imported API to appear in the component definition. References to elements of an imported API must be fully qualified unless they are imported by an import statement with a `from` clause. When a component imports a functional f (either a top-level function or a functional method) by an import statement with a `from` clause, the imported f may be overloaded with a functional f declared by the component. When a component imports a top-level declaration f from an API A , all the relevant types to type check the uses of f are implicitly imported from A . However, these implicitly imported types for type checking are not expressible by programmers; programmers must import the types explicitly by import statements to use them.

A key design choice we make is to require that components never refer to other components directly; all external references are to APIs. This requirement allows programmers to extend and test existing components more easily, swapping new implementations of libraries in and out of programs at will.

One important restriction on components is that no API may be both imported and exported by the same component. This restriction is required throughout to ground the semantics of operations on components, as discussed in Section 22.7.

Every component has a unique name, used for the purposes of component linking. This name includes a user-provided identifier. In the case of a simple component, the identifier is determined by a component name given at the top of the source file from which it is compiled. A build script may keep a tally on version numbers and append them to the first line of a component, incrementing its tally on each compilation. The name of a compound component is specified as an argument to the `link` operation (described in Section 22.7) that defines it.

Component equivalence is determined nominally to allow mutually recursive linking of components. By programmer convention, identifiers associated with components that are not included in the Fortress standard libraries begin with the reverse of the URL of the development team. A fortress does not allow the installation of distinct components with the same name. Component names are used during `link` and `upgrade` operations to ensure that the restrictions on upgrades to a component are respected, as explained in Section 22.7.

Every component also includes a vendor name, the name of the fortress it is compiled on, and a timestamp, denoting the time of compilation. The time of compilation is measured by the compiling fortress, and the name of the fortress is provided by the fortress automatically. Every timestamp issued by a fortress must be unique. The vendor name typically remains the same throughout a significant portion of the life of a user account, and is best provided as a user environment variable.

In our examples, we use published descriptions of packages in the Java 6.0 API [28] as examples of APIs expressible in our component system. We use, as names for these APIs, the names of the corresponding Java packages, with `java` replaced with `Fortress`. For example, the following is the beginning of a source file for a fictional application `IronCrypto`:

```
component Com.Sun.IronCrypto
import Fortress.IO
import Fortress.Security
```

```

export Fortress.Crypto
...
end

```

22.3 APIs

Syntax:

```

Api ::= api DottedId Imports? AbsDecls? end
AbsDecls ::= AbsDecl+
AbsDecl ::= AbsTraitDecl
           | AbsObjectDecl
           | AbsVarDecl
           | AbsFnDecl
           | DimUnitDecl
           | TypeAlias
           | TestDecl
           | PropertyDecl
           | AbsExternalSyntax

AbsTraitDecl ::= TraitHeader AbsGoInATrait? end
AbsGoInATrait ::= AbsGoFrontInATrait AbsGoBackInATrait?
                 | AbsGoBackInATrait
AbsGoFrontInATrait ::= AbsGoesFrontInATrait+
AbsGoesFrontInATrait ::= ApiFldDecl
                       | AbsGetterSetterDecl
                       | PropertyDecl
AbsGoBackInATrait ::= AbsGoesBackInATrait+
AbsGoesBackInATrait ::= AbsMdDecl
                       | AbsCoercion
                       | PropertyDecl
AbsObjectDecl ::= ObjectHeader AbsGoInAnObject? end
AbsGoInAnObject ::= AbsGoFrontInAnObject AbsGoBackInAnObject?
                   | AbsGoBackInAnObject
AbsGoFrontInAnObject ::= AbsGoesFrontInAnObject+
AbsGoesFrontInAnObject ::= ApiFldDecl
                          | AbsGetterSetterDecl
                          | PropertyDecl
AbsGoBackInAnObject ::= AbsGoesBackInAnObject+
AbsGoesBackInAnObject ::= AbsMdDecl
                          | AbsCoercion
                          | PropertyDecl
AbsCoercion ::= widening? coercion StaticParams? (Id IsType) CoercionClauses
ApiFldDecl ::= ApiFldMods? Id IsType
ApiFldMods ::= ApiFldMod+
ApiFldMod ::= hidden | settable | UniversalMod
AbsVarDecl ::= VarWTypes
               | VarWoTypes : TypeRef ...
               | VarWoTypes : SimpleTupleType

```

APIs are compiled from special API definitions. These are source files which declare the entities declared by the API, the names of all APIs referred to by those declarations, and prose documentation. In short, the source code of an API

should specify all the information that is traditionally provided for the published APIs of libraries in other languages.

The syntax of an API definition is identical to the syntax of a component definition, except that:

1. An API definition begins with `api` rather than `component`. As with components, the identifiers associated with APIs that are not included in the Fortress standard libraries are prefixed with the reverse of the URL of the development team.
2. An API does not include `export` statements. (However, it does include `import` statements, which name the other APIs used in the API definition.)
3. Only declarations (but not definitions!) are included in an API definition except `test` and `property` declarations. A method or field declaration may include the modifier `abstract`. (Whether a declaration includes the modifier `abstract` has a significant effect on its meaning, as discussed below).

For the sake of simplicity, every identifier reference in an API definition must refer either to a declaration in a used API (i.e., an API named in an `import` statement, or the Fortress core APIs, which are implicitly imported), or to a declaration in the API itself. In this way, APIs differ from signatures in most module systems: they are not parametric in their external dependencies.

Every API has a unique name that consists of a user-provided identifier. As with components, API equivalence is determined nominally. Every API also includes a vendor name, the name of the fortress it is compiled on, and a timestamp.

Component and API names exist in separate namespaces. For convenience, a compiler can also produce an API directly from a project with the same name as the component it is derived from. Such an API includes *matching* declarations of the component. All declarations in the component appear in the API.

For example, consider the APIs `Fortress.IO`, `Fortress.Security`, and `Fortress.Crypto`, with declarations similar to those in their respective Java packages. These APIs are interdependent. For example, both `PublicKey` in `Fortress.Security` and `SecretKey` in `Fortress.Crypto` extend the trait `Fortress.IO.Serializable` and the trait `CipherSpi` in `Fortress.Crypto` has methods that return values of type `AlgorithmParameters` in `Fortress.Security`. So the API `Fortress.Crypto` must import `Fortress.IO` and `Fortress.Security` as follows:

```
api Fortress.Crypto
import Fortress.IO
import Fortress.Security
...
end
```

22.4 Tests in Components and APIs

A component may include definitions of tests, as described in Chapter 19. These definitions are allowed to refer to both test and non-test code defined in the same component or declared in APIs imported by the component.

An API may also include definitions of tests. These definitions may refer to all declarations in the API as well as in any APIs it imports. Tests defined in APIs should be thought of as “executable documentation” that partially specifies the required behavior of the declared entities.

See Section 22.7 for an explanation of how tests defined in components and APIs are executed.

22.5 Type Inference for Components

Type inference for Fortress has been described as a procedure performed over a whole Fortress program in Chapter 20. In this section, we explain how this procedure can be adapted to perform type inference over a simple program component. For a compound component, concatenate all declarations of all constituents in the order specified by the constructing `link` operation. Constituent compound component declarations are recursively concatenated.

Type inference over a simple component C is performed by first expanding C into a self-contained Fortress program, as follows:

1. All program constructs corresponding to declarations in APIs exported by C are expanded so that they include all types and static parameters included in the exported APIs.
2. All types provided by all declarations in the APIs imported by C are prepended to C . Note that these declarations must include types for all variables, functions, fields, and methods; otherwise the APIs that declare them are not well-formed.
3. In order for the resulting expanded program to be well-formed, we assume that all declarations in these APIs are expanded into special definitions that include *empty bodies*. The empty body of such a definition is a conceptual body which cannot be expressed directly in Fortress programs. Because declarations in APIs do not have any elided type, type inference ignores empty bodies.

Once C is expanded, type inference is performed over all program constructs that still include elided types. Empty bodies are ignored.

22.6 Initialization Order for Components

To ensure that all objects and all variables are initialized before their use, execution of program components proceeds according to the procedure defined in this section. This procedure assumes that the program's type hierarchy is already checked to be acyclic.

If a component is a compound component, all constituent components are initialized nondeterministically, but before first use. If a simple component has imports, take the transitive closure of all imported APIs. Collect all declarations in this transitive closure, in any order, and prepend them to the component definition. Finally, for a simple component without imports, initialize all top-level variables and singleton object fields as described in Chapter 4.

22.7 Basic Fortress Operations

We now describe the operations that can be performed on a fortress by developers and end-users for developing, installing, testing, and maintaining components. We can think of these operations as commands to an interactive shell provided by the fortress.

In this section, we discuss operations on a fortress in their most basic form, postponing the discussion of more advanced options, including additional optional parameters, to Section 22.8. Although these more advanced options are critical to performing some real-world tasks with components, it is easier to describe their behavior after the basic forms of operations have been discussed.

Compile This operation takes the source code for a simple component (or API) definition and produces a new component object (or API object) that is installed on the fortress. Its type is as follows:

```
compile(file:String) : ()
```

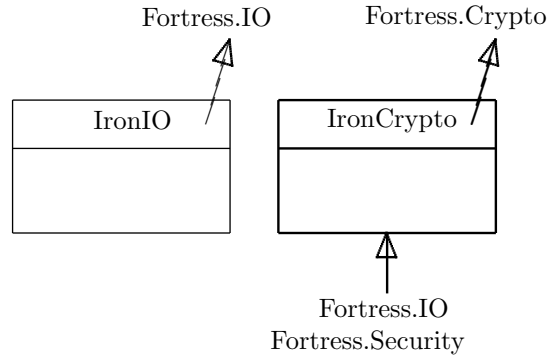


Figure 22.1: Simple components in box notation: A component is represented by a box, with the name of the component at the top of the box. The arrow protruding from the upper right corner of a box is labeled with the APIs exported by the component. The arrow pointing into the bottom of a box is labeled with APIs imported by the component. If no APIs are imported, we elide the arrow.

For example, suppose `IronCrypto.fss` contains the source code for the aforementioned `IronCrypto` application, which imports `Fortress.IO` and `Fortress.Security`, and exports `Fortress.Crypto`. Suppose we also have source code, `IronIO.fss`, for another application, `IronIO`, which imports nothing and exports `Fortress.IO`. We generate these components by compiling the source files:

```

compile("IronIO.fss")
compile("IronCrypto.fss")
  
```

The results are depicted diagrammatically in Figure 22.1.

Link A collection of one or more components exporting different APIs may be combined to form a new, compound, component by calling the `link` operation, passing the names of the components to link along with the name of the resulting compound component. Syntactically, a `link` operation is written as follows:⁴

```

link(result:String, constituents:String...):()
  
```

The components being linked are called *constituents* of the resulting component, which exports all the APIs exported by any of its constituents, and imports the APIs imported by at least one of its constituents but not exported by any of them.

For example, we can link the `IronIO` and `IronCrypto` libraries compiled above:

```

link(IronLink, IronIO, IronCrypto)
  
```

The resulting component, illustrated in Figure 22.2, imports `Fortress.Security` and exports `Fortress.IO` and `Fortress.Crypto`.

`link` does not distinguish between simple and compound components, so we can get arbitrarily nested components. For example, we can construct an application `CoolCryptoApp` by compiling another source code, `IronSecurity.fss`, for the library `IronSecurity` that imports `Fortress.IO` and exports `Fortress.Security`, and then linking the result with `IronLink`.

```

compile(IronSecurity.fss)
link(CoolCryptoApp, IronSecurity, IronLink)
  
```

The resulting components are illustrated in Figure 22.3.

⁴We present only the basic form of `link` here. `link` has additional optional arguments that we discuss in the Section 22.8.

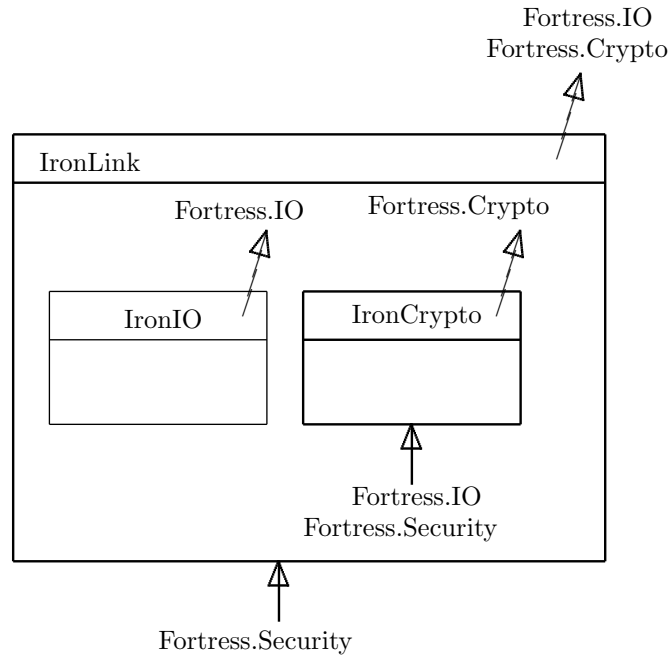


Figure 22.2: A compound component: A component inside another component is a constituent of the component that immediately encloses it.

Two components cannot be linked if they export the same API.⁵ This restriction is made for the sake of simplicity; it allows programmers to link a set of components without having to specify explicitly which constituent exporting an API *A* provides the implementation exported by the linked component, and which constituent connects to the constituents that import *A*: only one component exports *A*, so there is only one choice. Although we lose expressiveness with this design, the user interface to link is vastly simplified, and it is rare that including multiple components that export a given API in a set of linked components is even desirable. We discuss how even such rare cases can be supported in Section 22.8.

For a compound component, in addition to the exported and imported APIs, we want to know what its constituents are. It is an invariant of the system that for any compound component *C*, any API imported by any of its constituents is either imported by *C* or exported by one of its constituents. This property is crucial for executing components, as we discuss below. A simple component (i.e., one produced directly by compilation) has no constituents.

Execute Components provide implementations of the APIs they export. A component is *executable* if it imports no APIs and it exports the special API `Executable`, defined as follows:

```

api Executable
  run(args : String...): ()
end
  
```

An executable component may be *executed* by calling the `execute` operation, resulting in a call to the component's implementation of the `run` function in a new process. Arguments to the `run` function are passed to the shell:

```

execute(componentName:String, args:String...): ()
  
```

We say that a component is being executed when `execute` has been called on that component and has not yet returned, or if it is the constituent component of a component being executed. During an execution, references may be made to

⁵There is one exception to this rule: the special API `Upgradable`, which is used during upgrades discussed below.

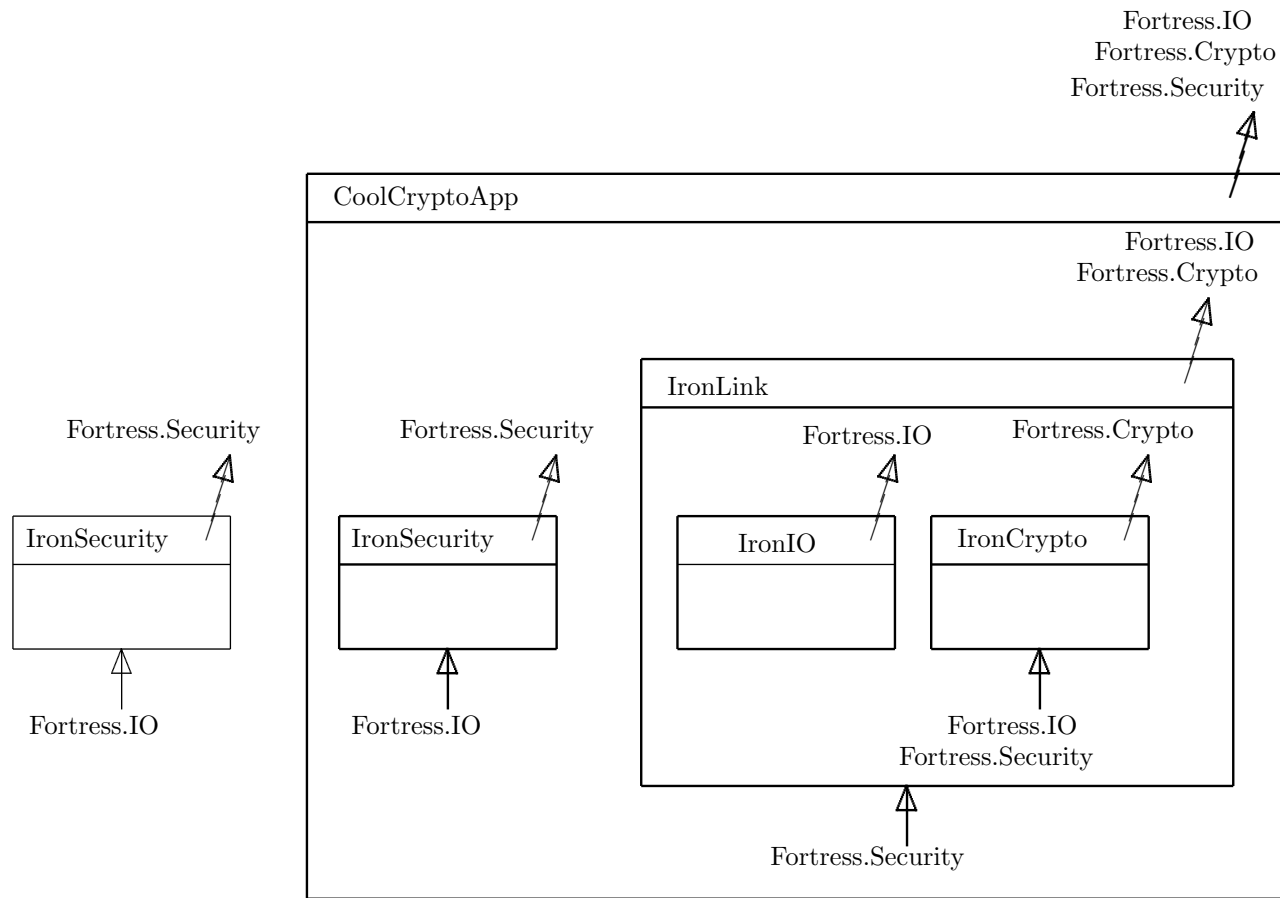


Figure 22.3: Repeated linking

APIs exported by a component being executed, which may in turn make references to APIs that it imports.

For references to an API *A* exported by the component, if the component is simple, then it contains the code necessary to evaluate any reference to an API it exports, possibly making references to APIs that it imports to do so. If the component is compound, then it contains a unique constituent that exports *A*; the reference is resolved to that constituent component.

For external references within a constituent component, recall that all such references in a component must be to APIs that the component imports. A component being executed either does not import any API (and thus there are no external references to resolve), or else is a constituent of another component that is being executed. In the latter case, the constituent defers the reference to its enclosing component.

For example, suppose `CoolCryptoApp` above is the constituent of some executable component, and when that component is executed, it generates a reference to `SecretKey` in `Fortress.Crypto`, which it resolves to `CoolCryptoApp`. `CoolCryptoApp` resolves this reference to `IronLink`, which resolves it to `IronCrypto`, which is a simple component. Suppose that in evaluating this reference, `IronCrypto` generates a reference to `PublicKey` in `Fortress.Security`. Because `IronCrypto` imports `Fortress.Security`, it resolves this reference to its enclosing component, `IronLink`, which in turn resolves it to `CoolCryptoApp`, which resolves it to `IronSecurity`, which is a simple component.

Not all projects are compiled to components that export `Executable`. For example, a library component does not usually export `Executable`.

Upgrade Compound components may be upgraded with new constituent components by calling an `upgrade` operation, passing the name of the component to upgrade (the *target*), the name of a component to upgrade with (the *replacement*), and a name for the resulting component (which we call the *result*). The type of the `upgrade` operation is as follows:

```
upgrade(target:String, replacement:String, result = target):()
```

If no result name is provided, the result is bound to the name of the target, and the target is uninstalled (see below).

For example, we can upgrade `CoolCryptoApp` with a component `CoolSecurity`, which exports `Fortress.Security` and imports nothing to `CoolCryptoApp.2.0`.

```
upgrade(CoolCryptoApp, CoolSecurity, CoolCryptoApp.2.0)
```

The resulting component is illustrated in Figure 22.4. Notice that the constituent, `IronSecurity`, exporting `Fortress.Security` has been replaced.

A component can be upgraded only if it exports the special API `Upgradable`, defined as follows:

```
api Upgradable
import Component from Components
isValidUpgrade(that : Component) : Boolean
upgrade(that : Component) : Component requires { isValidUpgrade(that) }
end
```

The `Upgradable` API imports a special API `Components` that provides handles on `Component` and `Api` objects. The `Components` API is described in Chapter 39.

An `upgrade` operation on a component invokes the `isValidUpgrade` function, as declared in the API `Upgradable`. This function must take a component and return `true` if and only if it is legal to upgrade with respect to that component. Developers can define their own versions of this component to restrict how their components can be upgraded. For example, they can prevent upgrades with older versions of a component, or with a matching component from an untrusted vendor.

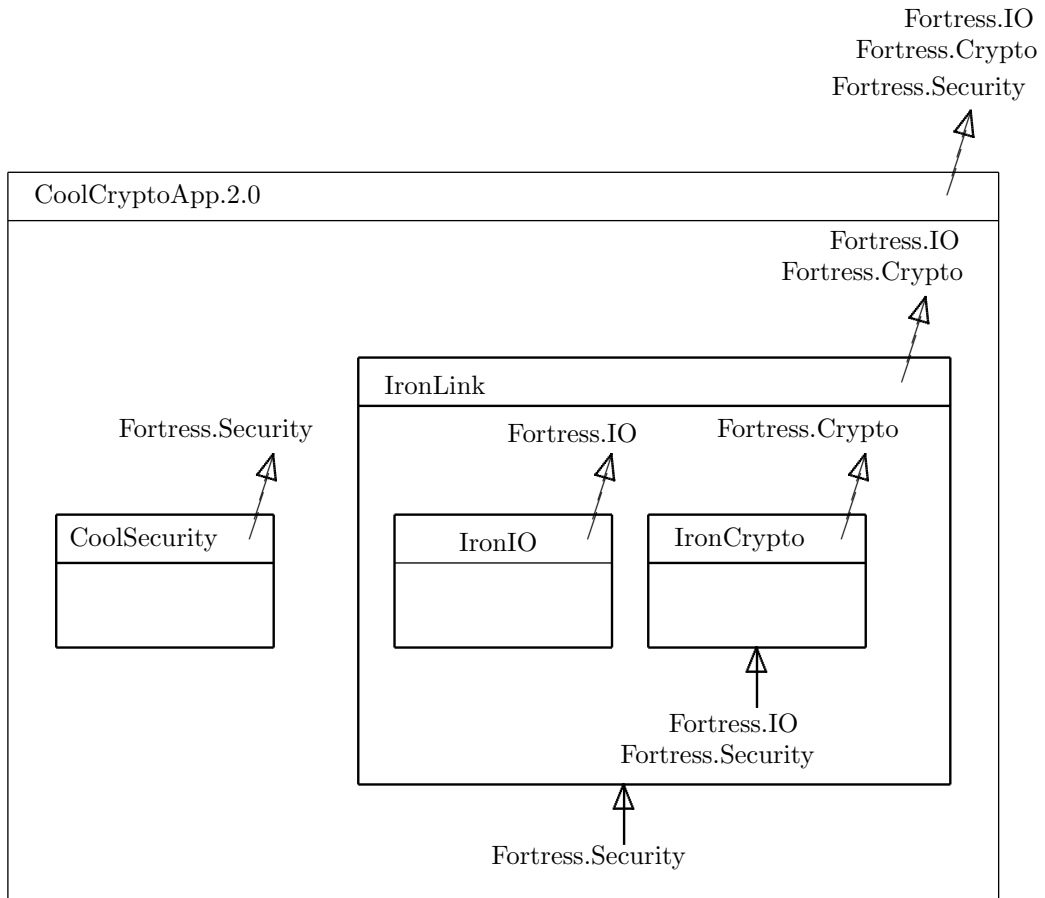


Figure 22.4: An upgraded component

The Upgradable API presents a problem for our model. Its implementation by the various constituent components in a compound component must be accessed during an `upgrade` operation. However, because the exported APIs of the constituent components must be disjoint, they cannot all export `Upgradable` after linking.

We solve this problem by introducing an additional step during linking. In a `link` operation, a special component, called a *restriction component*, is constructed automatically, based on the provided constituents. This component exports the `Upgradable` API; its implementation is a function of all the constituents provided to the `link` operation. The provided constituents are then used to construct a new set of constituents that are identical to the provided constituents except that they do not export `Upgradable`. These new constituents are then combined, along with the restriction component, to form the constituents of a new compound component.

In addition to the constraints imposed by a component's *isValidUpgrade* function, there are several other conditions that must be met in order for an upgrade to be valid. These conditions are necessary to ensure that the resulting component is well-formed and imports and exports the same APIs as the target: ⁶

1. Every API imported by the replacement must be either imported or exported by the target.
2. The APIs exported by the replacement must be a subset of those exported by the target.
3. If the replacement does not export all the APIs that a constituent exports then either the replacement and constituent do not export any APIs in common or the constituent can be upgraded with the replacement.

The rationale for the first two conditions is straightforward: If an API is imported by the replacement but not imported or exported by the target, then references to that API cannot be resolved in the result (unless we also import that API in the result). If an API is exported by the replacement but not the target, then the result will export an API not exported by the target.

The third condition says that the constituents of the target can be partitioned into three sets: those that are subsumed by the replacement, those that are unaffected by the upgrade, and all the rest, which can be upgraded with the replacement. This condition enables recursive propagation of upgrades. That is, an upgrade not only replaces constituents at the top level of the component, but is also propagated into any constituents with which it exports some APIs in common. Thus, in the example above, we could have upgraded `CoolCryptoApp` with a component that exports `Fortress.IO`. However, we could not have upgraded `CoolCryptoApp` with a component that exports both `Fortress.Security` and `Fortress.IO` because `IronLink` exports `Fortress.IO` but not `Fortress.Security`. In Section 22.8, we show how hiding and constraining APIs can help us get around many of the limitations that this condition imposes.

Recall that in our system, unlike with dynamic linking, components are encapsulated so that an upgrade to one component does not affect any other component on the system. We can imagine that all operations on components copy the components that they operate on rather than share them. Because components are immutable, these two interpretations are semantically indistinguishable. Convenience operations that support mass upgrades are provided on fortresses (e.g., an `upgradeAll` operation that takes a component and upgrades all components in the fortress that can be upgraded with its argument).

Extract and Install A component installed on a fortress may be *extracted* by calling an `extract` operation on the fortress, passing the name of the component as an argument, along with an argument `prereqs`, denoting the names of all APIs that must be installed on any fortress before this component can be installed.

```
extract(componentName:String, prereqs:Set[String] = {}):()
```

Furthermore, the destination fortress must have a component that exports these APIs and is a valid upgrade of the extracted component. Intuitively, a `prereqs` argument allows a component to be serialized without having to include all of its libraries; new libraries can be provided when the component is installed at a destination fortress.

The `prereqs` argument is optional; if omitted, the extracted component can be installed on any fortress. Any component can be extracted; however only compound components can be extracted with a `prereqs` argument: because

⁶ These conditions are sufficient provided there are no hidden or constrained APIs, which are discussed in Section 22.8.

extracted components must be upgradable with respect to a component exporting their `prereqs`, no `prereqs` argument makes sense for a simple component.

The APIs included in a `prereqs` argument must be the APIs exported by some subset of the extracted component's constituents (or a subset of the constituents of one of its constituents, and so on, due to recursive updating).

The extracted component is serialized to a file, including all the APIs it refers to (and, transitively, all APIs they refer to) and all constituent components, except those that export the `prereqs`. This operation does not remove the extracted component from the fortress; there is a separate `uninstall` operation for that.

When the component is extracted, if no `prereqs` were passed to the `extract` operation, then the contents of the file can be deserialized by any fortress into the extracted component, which can be installed on the fortress. However, if `prereqs` were passed to `extract`, then the file must be deserialized into a component that exports only the Installable API:

```
api Installable
import Component from Components
reconstitute(candidate : Component) : Component
end
```

The deserialized component is immediately linked with preferred implementations of all of its imported APIs. (Preferred implementations of APIs are maintained in a table by a fortress, which maps each API to a list of components that implements it, in order of preference). Because the deserialized and linked component exports the Installable API, it has a `reconstitute` function that takes a `candidate` component, which exports the `prereqs` APIs, and checks whether the given component satisfies the `isValidUpgrade` condition of the extracted component. If so, it returns the extracted component upgraded with the given component. The `reconstitute` function is called by the fortress with a new component, formed by linking the preferred components for each API in the extracted components' `prereqs` argument.

Note that an extracted component with `prereqs` APIs is *not* the same as an extracted component that imports the same APIs but has no `prereqs` APIs. The latter can always be installed on a fortress, and then can be subsequently linked with any component that exports the imported APIs. In contrast, the fortress has no access to an extracted component with `prereqs` APIs unless it has a component that exports these APIs and satisfies the `isValidUpgrade` function of the extracted component. This difference provides a means for controlling access to the extracted component, for security, legal, or other reasons.

Syntactically, an `install` operation takes the name of a file constraining an extracted component. The `install` operation is overloaded with another operation that takes the name of a component to match `prereqs`. If this optional argument is provided, and the deserialized component exports the Installable API, then the `reconstitute` function is called with the component denoted by the optional argument of `install`, rather than the fortress' preferred implementation of the `prereqs` APIs. Install operations are written as follows:

```
install(file:String) : ()
install(file:String, prereqs:Set[\String\]) : ()
```

By default, a fortress adds a newly installed component to the head of the "preferred" list for every API it exports. However, this default may be overridden by the end-user; an end-user may modify the table or even map some APIs differently during a particular installation. If one or more of the APIs required by an extracted component is not mapped to an API on the destination fortress, an exception is thrown.

There is a corresponding operation for APIs, `installAPI`, that takes a serialization of a set of APIs and installs them into a fortress.

```
installAPI(file:String) : ()
```

This set of APIs must be closed under imports. If an API that is installed in this way is already installed on the fortress, the definitions must match exactly, or an exception is thrown.

Uninstall An `uninstall` operation takes the name of a component as an argument and removes the top-level binding of that component from a fortress. Note that the uninstalled component may have been linked to other components, or used as a replacement in an upgrade, and the result may still be installed; an `uninstall` operation will not affect these other components.

```
uninstall(file:String):()
```

There is a corresponding operation for APIs, `uninstallAPI`, that removes an API from a fortress.

```
uninstallAPI(file:String):()
```

Typically, this operation is used only to remove APIs that have been corrupted in some fashion.

Testing A component can be tested by calling the method `runTests` on it:

```
runTests(inclusive = true):()
```

This method runs all test functions defined in the component. All test functions are run in parallel; each test function is run for each combination of test cases (bound in its generator list as described in Section 19.2) in parallel. In the case of a compound component, the set of defined test functions consists of all test functions defined by all constituent components and by all exported APIs. The set of test functions run can be limited by first hiding the tested component in a more restrictive API. The set of test functions can also be expanded by linking with a component defining additional test functions.

The `runTests` method includes a keyword parameter `inclusive` that defaults to `true`. If this parameter is set to `false`, only test functions defined in the APIs exported by the component are run.

22.8 Advanced Features of Fortress Operations

The system we have described thus far provides much of the desired functionality of a component system. However it has a few significant weaknesses:

1. It exposes to everyone all the APIs used in the development of a project.
2. By allowing access to these APIs, it inhibits significant cross-component optimization.
3. It prevents components that use two different implementations of the same API from being linked, even if they never actually pass references to that API between each other.
4. It restricts the upgradability of compound components, as described earlier.

We can mitigate all these shortcomings by providing two simple operations, `hide` and `constrain`. Informally, `hide` makes APIs no longer visible from outside the component so that they cannot be upgraded, and `constrain` merely prevents them from being exported. An API that is constrained but not hidden can still be upgraded. There are other subtle consequences of this distinction, which we discuss as they arise.

Some of the properties about the APIs exported by a component discussed in Section 22.7 are actually properties of APIs that are visible or provided by a component. For example, APIs visible in a component cannot be imported by that component, even if they are not exported. Other properties are really properties only of the exported APIs. Most importantly, components that do not export any common APIs can be linked, as can components that share only visible APIs.

Constrain A `constrain` operation takes a component name of an installed component, a new component name, and a set of APIs, and produces a new component that does not export any of the APIs specified. Syntactically, we write:

```
constrain(source:String, destination = source, apis:Set[\String\]):()
```

If no destination name is provided, the name of the `source` is used.

The set of APIs provided must be a subset of the APIs exported by the component. Also, recall that every API used by an API exported by a component must be imported or exported by that component. Thus, if we constrain an API that is used by any other API exported by the component, then we must also constrain that other API.

If the component is a simple component, we first link it by itself, and then apply `constrain` to the result.

Hide A `hide` operation is like a `constrain` operation, except that the given set of APIs is subtracted from the visible and provided APIs, along with the exported APIs, in the resulting component.

```
hide(source:String, destination = source, apis:Set[\String\]):()
```

The requirement of APIs being imported or exported whenever an API using them is exported also applies to visible APIs. Thus, if we hide an API used by another exported API, we must hide that other API as well.

Link With constrained APIs, there is a new restriction on link: Any API visible in one constituent and imported by another must be exported by some constituent. This restriction is necessary because an API visible in a component cannot be imported by that component. Thus, if one of the component's constituents imports that API, then the API must be provided by some other constituent. Other than that, the `link` operation is largely unchanged: the visible APIs are just all the APIs visible in any constituent, and the provided APIs are just those exported by any constituent. There is a subtle additional restriction on how linked components can be upgraded, which we discuss below.

Rather than requiring users and developers to call `constrain` and `hide` directly, we provide optional parameters to the `link` operation to do these operations immediately. The `link` operation has the following type:

```
link(result:String, constituents:String..., exports = {}, hide = {}):()
```

If the `exports` clause is present, only those APIs listed in the set following `exports` are exported; the others are constrained. If the `hide` clause is present, those APIs listed in the set following `hide` are hidden. An exception is thrown if the `exports` clause contains any API not exported by any constituent, or if the `hide` clause contains any API not visible in any constituent.

Hiding enables us to handle the rare case in which programmers want to link multiple components that implement the same API without upgrading them to use the same implementation. Before linking, the programmer simply hides (or constrains) the API in every component that exports it except the one that should provide the implementation for the new compound component.

For example, suppose we wish to link the following two components:

- A component `NetApp` that imports `Fortress.IO` and exports the `Fortress.Net` API.
- A component `EditApp` that imports `Fortress.IO` and exports the `Fortress.Swing.Textrf` API.

We want to link these two components to use in building an application for editing messages and sending them over a network. But we want to use different implementations of `Fortress.IO` (e.g., `IOApp1` and `IOApp2` for the two components). We simply perform the following operations:

```
link(temp1, NetApp, IOApp1, exports = {Fortress.Net}, hide = {Fortress.IO})
link(temp2, EditApp, IOApp2, exports = {Fortress.Swing.Textrf}, hide = {Fortress.IO})
link(NetEdit, temp1, temp2)
```

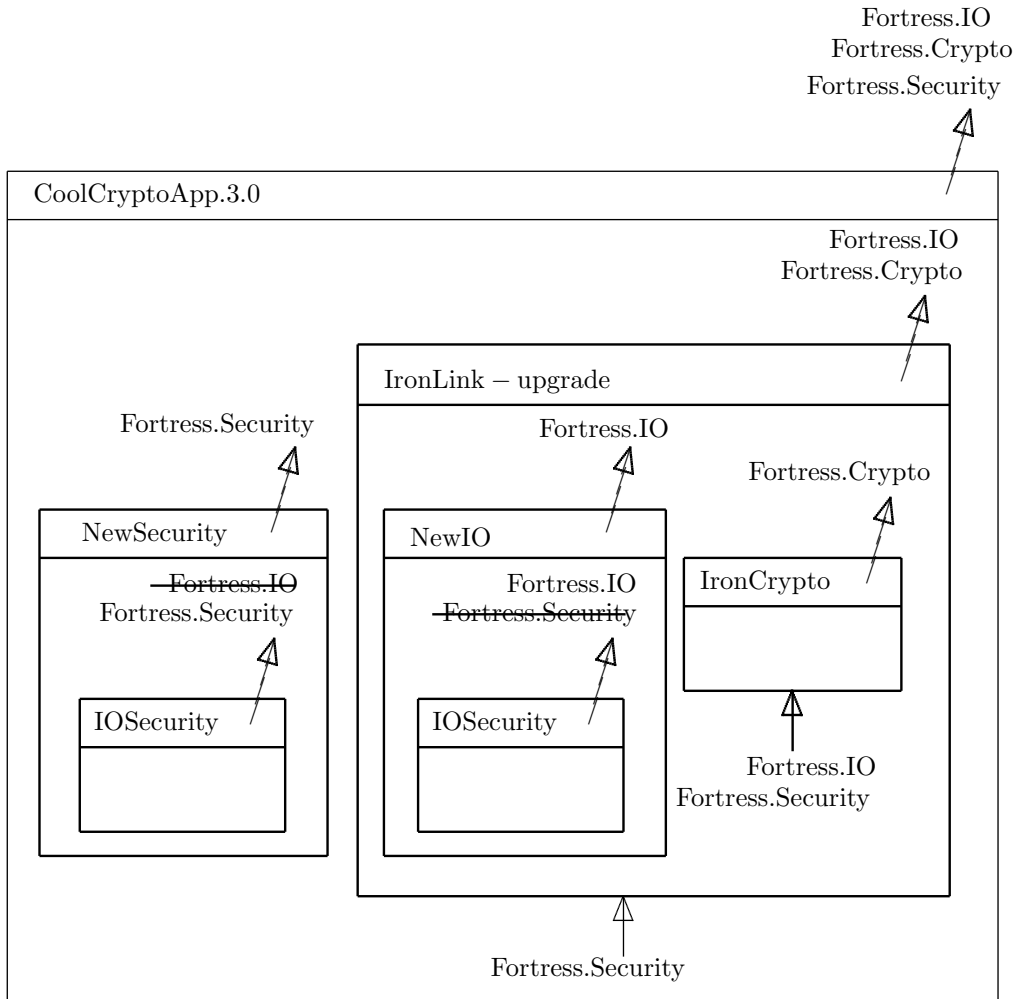


Figure 22.5: Upgrading with hidden APIs: Crossed out APIs are hidden.

In this case, the NetEdit component does not export, or even make visible, Fortress.IO at all.

Upgrade For the `upgrade` operation, there is no change at all in the semantics. However, because hiding and constraining APIs allow us to change the APIs exported by a component, it is possible to do some upgrades that are not possible without these operations.

For example, suppose we have a component `IOSecurity` that exports `Fortress.IO` and `Fortress.Security`, and we want to upgrade `CoolCryptoApp` with `IOSecurity`. As discussed above, we cannot use `IOSecurity` directly because `IronLink` exports `Fortress.IO` but not `Fortress.Security`. We can get around this restriction by doing two upgrades, one with `Fortress.Security` hidden and the other with `Fortress.IO` hidden.

```
hide(IOSecurity, NewIO, Fortress.Security)
hide(IOSecurity, NewSecurity, Fortress.IO)
upgrade(CoolCryptoApp, NewSecurity, temp1)
upgrade(CoolCryptoApp.3.0, temp1, NewIO)
```

The resulting component is shown in Figure 22.5.

The interplay between imported, exported, visible and provided APIs introduces subtleties that not present in our

discussion above. In particular, the last of the three conditions imposed for well-formedness of upgrades is modified to state that for any constituent that is not subsumed by a replacement component, either it can be upgraded with the replacement, or its *visible* APIs are disjoint from the APIs exported by the replacement (i.e., it is unaffected by the upgrade). To maintain the invariant that no two constituents export the same API, we need another condition, which was implied by the previous condition when no APIs were constrained or hidden: if the replacement subsumes any constituents of the target, then its exported APIs must exactly match the exported APIs of some subset of the constituents of the target. In practice, this restriction is rarely a problem; in most cases, a user wishes to upgrade a target with a new version of a single constituent component, where the APIs exported by the old and new versions are either an exact match, or there are new APIs introduced by the new component that have no implementation in the target.

Part III

Fortress APIs and Documentation for Application Programmers

Chapter 23

Objects

23.1 The Trait `Fortress.Core.Any`

The trait `Any` is the single root of the type hierarchy; every object in Fortress has trait `Any` and therefore every object implements the methods of this trait. The immediate subtypes of the trait `Any` are `Object`, `Tuple`, and `()`.

```
trait Any extends { }
  opr  $\equiv$ (self, other: Any): Boolean
  opr IDENTITY(self): Any
  hash(maxval: N64): N64
  hash(maxval: N32): N32
  getter hashCode(): N64
  toString(): String
  property  $\forall(x, y, n: N64) x \equiv y \rightarrow x.hash(n) \equiv y.hash(n)$ 
  property  $\forall(x, y, n: N32) x \equiv y \rightarrow x.hash(n) \equiv y.hash(n)$ 
  property  $\forall(x) x.hashCode \equiv x.hash(2^{64} - 1)$ 
  property  $\forall(x, y) x \equiv y \rightarrow x.toString() = y.toString()$ 
  property  $\forall(a) (IDENTITY a) \equiv a$ 
  property  $\forall(a) a \equiv a$ 
  property  $\forall(a, b) a \equiv b \leftrightarrow b \equiv a$ 
  property  $\forall(a, b, c) (a \equiv b \wedge b \equiv c) \rightarrow a \equiv c$ 
end
```

23.1.1 `opr \equiv (self, other: Any): Boolean`

The infix operator `\equiv` (object equivalence) is used to decide whether two objects are “the same object” in the strictest sense possible; this is described in detail in Section 10.4. Note that the properties of trait `Any` assert that `\equiv` is an equivalence relation. In `Object` this property is asserted abstractly using the algebraic constraints of Chapter 37.

For `\neq` see Section 26.1.1.

23.1.2 `opr IDENTITY(self): Any`

The operator `IDENTITY` simply returns its argument. (This may not be terribly useful for applications programming, but it has technical uses for specifying contracts and algebraic properties in libraries as described in Section 37.3.)

23.1.3 `hash(maxval: N64): N64`

23.1.4 `hash(maxval: N32): N32`

The `hash` method returns a *hash value* for the object as an unsigned integer that is less than or equal to the `maxval` argument. This hash value is not necessarily consistent from one Fortress application to another, nor from one execution of a Fortress application to another execution of the same application, but the hash value produced for a given value of the `maxval` argument remains fixed during the execution of a single Fortress application. There is no defined relationship between hash values produced for the same object but with different `maxval` values. Fortress programmers and implementors should be aware that the performance of hash tables is likely to be improved if, for any given collection of objects and given value for the `maxval` argument, the `hash` method assigns hash values to those objects with relatively uniform distribution.

23.1.5 `getter hashCode(): N64`

Every object has associated with it a 64-bit unsigned integer value called its *hash code*; this is the value returned by the `hash` method when given the argument $2^{64} - 1$. Hash codes are not necessarily consistent from one Fortress application to another, nor from one execution of a Fortress application to another execution of the same application, but remain fixed during the execution of a single Fortress application. It is permitted for two objects to have the same `hashCode`, but Fortress programmers and implementors should be aware that assigning distinct hash codes to distinct objects may improve the performance of hash tables.

The trait `Any` defines its `hash` methods in terms of `hashCode`; therefore it suffices for a subtrait to override `hashCode` to get the benefit of the `hash` methods as well.

23.1.6 `toString(): String`

The general contract of `toString` is that it returns a string that “textually represents” this object. The idea is to provide a concise but informative representation that will be useful to a person reading it.

23.2 The Trait `Fortress.Core.Object`

The trait `Object` is the root of the type hierarchy for all user-constructed objects and functions. The `Object` type does not have any additional methods, but uses algebraic constraints (see Chapter 37) to describe the properties of the existing operators more abstractly.

```
trait Object extends { Any, EquivalenceRelation[[Object, ≡]], IdentityOperator[[Object]] }
  excludes { Tuple }
end
```

23.3 The Trait Fortress.Core.Tuple

The trait `Tuple` is the root of the type hierarchy for all tuple types (see Section 8.4). No user-defined object can be a subtype of `Tuple`. As with the type `Object`, the type `Tuple` uses algebraic constraints to describe its existing methods.

```
trait Tuple extends { Any, EquivalenceRelation[[Tuple, ≡]], IdentityOperator[[Tuple]] }
  excludes { Object }
  toString(): String
end
```

23.3.1 `toString(): String`

The `toString` method returns a string “” concatenated as follows:

- If the tuple has just one element, concatenate “tuple”.
- Concatenate “(”.
- For each element of the tuple, in left-to-right order, taking all plain elements before any varargs or keyword element, taking any varargs element before any keyword element, and taking keyword elements in the order defined by the type of the tuple (which may not be the sorted order):
 - If this is a plain element, concatenate the `toString` of the element.
 - If this is a varargs element “*e* . . .”, concatenate the `toString` of *e* and concatenate “. . .”.
 - If this is a keyword element “*id* = *e*”, concatenate the `toString` of *id*, concatenate “ = ”, and concatenate the `toString` of *e*.
 - If this is the last element of the tuple, concatenate “)”; otherwise concatenate “,”.

Chapter 24

Booleans and Boolean Intervals

24.1 The Trait `Fortress.Core.Boolean`

```
trait Boolean
  extends { BooleanAlgebra[[Boolean, juxtaposition, ∨, ¬, √]],
           BooleanAlgebra[[Boolean, juxtaposition, ∨, ¬, ⊕]],
           BooleanAlgebra[[Boolean, ∧, ∨, ¬, √]],
           BooleanAlgebra[[Boolean, ∧, ∨, ¬, ⊕]],
           BooleanAlgebra[[Boolean, ∨, juxtaposition, ¬, ≡]],
           BooleanAlgebra[[Boolean, ∨, juxtaposition, ¬, ↔]],
           BooleanAlgebra[[Boolean, ∨, ∧, ¬, ≡]],
           BooleanAlgebra[[Boolean, ∨, ∧, ¬, ↔]],
           TotalOrder[[Boolean, →]],
           PartialOrderAndBoundedLattice[[Boolean, →, juxtaposition, ∨]],
           PartialOrderAndBoundedLattice[[Boolean, →, ∧, ∨]],
           IdentityEquality[[Boolean]],
           EquivalenceRelation[[Boolean, ≡]],
           EquivalenceRelation[[Boolean, ↔]],
           Symmetric[[Boolean, ∧]], Symmetric[[Boolean, ∨]],
           Symmetric[[Boolean, √]], Symmetric[[Boolean, ⊕]],
           Symmetric[[Boolean, ¬]], Commutative[[Boolean, ¬]],
           Symmetric[[Boolean, ∇]], Commutative[[Boolean, ∇]] }
  comprises { ... }
  coercion [[bool b]](x: BooleanLiteral[[b]])
  coercion (x: Identity[[juxtaposition]])
  coercion (x: Identity[[∧]])
  coercion (x: Identity[[∨]])
  coercion (x: Identity[[√]])
  coercion (x: Identity[[⊕]])
  coercion (x: Identity[[≡]])
  coercion (x: Identity[[↔]])
  coercion (x: ComplementBound[[∧]])
  coercion (x: ComplementBound[[∨]])
  coercion (x: Zero[[∧]])
```

```

coercion (x: Zero[ $\forall$ ])
coercion (x: MaximalElement[ $\rightarrow$ ])
coercion (x: MinimalElement[ $\rightarrow$ ])
opr juxtaposition (self, other: Boolean): Boolean
opr  $\wedge$ (self, other: Boolean): Boolean
opr  $\wedge$ (self, other: ()  $\rightarrow$  Boolean): Boolean
opr  $\vee$ (self, other: Boolean): Boolean
opr  $\vee$ (self, other: ()  $\rightarrow$  Boolean): Boolean
opr  $\neg$ (self): Boolean
opr  $\underline{\vee}$ (self, other: Boolean): Boolean
opr  $\oplus$ (self, other: Boolean): Boolean
opr  $\equiv$ (self, other: Boolean): Boolean
opr  $=$ (self, other: Boolean): Boolean
opr  $\leftrightarrow$ (self, other: Boolean): Boolean
opr  $\rightarrow$ (self, other: Boolean): Boolean
opr  $\rightarrow$ (self, other: ()  $\rightarrow$  Boolean): Boolean
opr  $\overline{\wedge}$ (self, other: Boolean): Boolean
opr  $\overline{\vee}$ (self, other: Boolean): Boolean
opr  $\equiv$ (self, other: Boolean): Boolean
majority(self, other1: Boolean, other2: Boolean)
getter hashCode():  $\mathbb{N}64$ 
toString(): String
property  $\forall(a, b) a b = (a \wedge b)$ 
property  $\forall(a, b) (a \underline{\vee} b) = (a \oplus b)$ 
property  $\forall(a, b) (a \equiv b) = (a \leftrightarrow b) = (a = b) = (a \equiv b)$ 
property  $\forall(a, b) (a \rightarrow b) = ((\neg a) \vee b)$ 
property  $\forall(a, b) (a \overline{\wedge} b) = \neg(a \wedge b)$ 
property  $\forall(a, b) (a \overline{\vee} b) = \neg(a \vee b)$ 
end
test testData[] = { false, true }

```

24.1.1 coercion $\llbracket \text{bool } b \rrbracket(x: \text{BooleanLiteral}\llbracket b \rrbracket)$

A boolean literal can always serve as a Boolean value.

- 24.1.2** coercion (x : Identity[[juxtaposition]])
- 24.1.3** coercion (x : Identity[[\wedge]])
- 24.1.4** coercion (x : Identity[[\vee]])
- 24.1.5** coercion (x : Identity[[$\underline{\vee}$]])
- 24.1.6** coercion (x : Identity[[\oplus]])
- 24.1.7** coercion (x : Identity[[\equiv]])
- 24.1.8** coercion (x : Identity[[\leftrightarrow]])
- 24.1.9** coercion (x : ComplementBound[[\wedge]])
- 24.1.10** coercion (x : ComplementBound[[\vee]])
- 24.1.11** coercion (x : Zero[[\wedge]])
- 24.1.12** coercion (x : Zero[[\vee]])
- 24.1.13** coercion (x : MaximalElement[[\rightarrow]])
- 24.1.14** coercion (x : MinimalElement[[\rightarrow]])

The identity for \wedge or juxtaposition is *true*; the zero and the complement bound for \wedge or juxtaposition are *false*.

The identity for \vee is *false*; the zero and the complement bound for \vee are *false*.

The identity for $\underline{\vee}$ or \oplus is *false*; the identity for \equiv or \leftrightarrow is *true*.

The maximal element for \rightarrow (regarded as a partial order operator) is *true*, and the minimal element is *false*.

- 24.1.15** opr juxtaposition (self, other: Boolean): Boolean

Juxtaposition of boolean expressions is equivalent to using the logical AND operator \wedge .

- 24.1.16** opr \wedge (self, other: Boolean): Boolean
- 24.1.17** opr \wedge (self, other: () \rightarrow Boolean): Boolean

The logical AND operator \wedge (AND) returns *true* if both arguments are *true*; otherwise it returns *false*.

The conditional logical AND operator \wedge : (AND:) examines its first argument; if it is *false*, the result is *false*, and the second argument (a thunk) is not evaluated. But if the first argument is *true*, the second argument is evaluated and its result becomes the result of the conditional logical AND operator expression.

- 24.1.18** opr \vee (self, other: Boolean): Boolean
- 24.1.19** opr \vee (self, other: () \rightarrow Boolean): Boolean

The logical OR operator \vee (OR) returns *false* if both arguments are *false*; otherwise it returns *true*.

The conditional logical OR operator \vee : (OR:) examines its first argument; if it is *true*, the result is *true*, and the second argument (a thunk) is not evaluated. But if the first argument is *false*, the second argument is evaluated and its result becomes the result of the conditional logical OR operator expression.

24.1.20 `opr` \neg (`self`): Boolean

The logical NOT operator \neg (NOT) returns *true* if its argument is *false*; it returns *false* if its argument is *true*.

24.1.21 `opr` $\underline{\vee}$ (`self`, `other`: Boolean): Boolean

24.1.22 `opr` \oplus (`self`, `other`: Boolean): Boolean

The logical exclusive OR operator $\underline{\vee}$ (XOR) returns *true* if the arguments are different, one being *true* and the other *false*; it returns *false* if both arguments are *true* or both arguments are *false*.

The operator \oplus (OPLUS) does the same thing as $\underline{\vee}$.

24.1.23 `opr` \equiv (`self`, `other`: Boolean): Boolean

24.1.24 `opr` $=$ (`self`, `other`: Boolean): Boolean

24.1.25 `opr` \leftrightarrow (`self`, `other`: Boolean): Boolean

The logical equivalence, or exclusive NOR, operator \equiv (EQV) returns *true* if both arguments are *true* or both arguments are *false*; it returns *false* if the arguments are different, one being *true* and the other *false*. (Thus its behavior on boolean values happens to be exactly the same as that of the strict equivalence operator \equiv .)

The equality operator $=$ and the if-and-only-if operator \leftrightarrow (IFF) do the same thing as \equiv .

For \neq see Section 26.1.2. For \neq see Section 26.1.4.

24.1.26 `opr` \rightarrow (`self`, `other`: Boolean): Boolean

24.1.27 `opr` \rightarrow (`self`, `other`: ()) \rightarrow Boolean): Boolean

The logical implication operator \rightarrow (IMPLIES) returns *false* if the first argument is *true* but the second argument is *false*; otherwise it returns *true*.

The conditional logical implication operator \rightarrow : (IMPLIES:) examines its first argument; if it is *false*, the result is *true*, and the second argument (a thunk) is not evaluated. But if the first argument is *true*, the second argument is evaluated and its result becomes the result of the conditional logical implication operator expression.

24.1.28 `opr` $\bar{\wedge}$ (`self`, `other`: Boolean): Boolean

The logical NAND (NOT AND) operator $\bar{\wedge}$ (NAND) returns *false* if both arguments are *true*; otherwise it returns *true*.

24.1.29 `opr` $\bar{\vee}$ (`self`, `other`: Boolean): Boolean

The logical NOR (NOT OR) operator $\bar{\vee}$ (NOR) returns *false* if both arguments are *false*; otherwise it returns *true*.

24.1.30 `opr` \equiv (`self`, `other`: Boolean): Boolean

Two boolean values are strictly equivalent if and only if they are the same boolean value (that is, both *true* or both *false*).

24.1.31 `majority`(`self`, `other1`: Boolean, `other2`: Boolean)

If two or three of the arguments are *true*, the result is *true*; if two or three of the arguments are *false*, the result is *false*.

24.1.32 `getter hashCode`(): $\mathbb{N}64$

24.1.33 `toString`(): String

The `toString` method returns either “true” or “false” as appropriate.

24.2 The Trait `Fortress.Standard.BooleanInterval`

A boolean interval is a set of boolean values. There are two distinct boolean values, *true* and *false*, and therefore there are four distinct boolean intervals, which for convenience are given names:

$$\begin{aligned}\text{True} &= \{true\} \\ \text{False} &= \{false\} \\ \text{Uncertain} &= \{true, false\} \\ \text{Impossible} &= \{\}\end{aligned}$$

Logical operations on intervals obey the interval containment rule: the result interval must contain every boolean result that can be produced by applying the operator to a boolean value taken from each argument interval. For example, if P and Q are boolean intervals, then by definition $P \wedge Q = \{x \wedge y \mid x \leftarrow P, y \leftarrow Q\}$.

A principal application of boolean intervals is to express the results of numerical comparison of numerical intervals. In this way numerical comparisons can also obey the interval containment rule.

Set operations such as \cup and \cap may also be used on boolean intervals.

```

trait BooleanInterval
  extends { BooleanAlgebra[[BooleanInterval,  $\cap$ ,  $\cup$ , SET_COMPLEMENT, SYMDIFF]],
    Set[[Boolean]],
    BinaryIntervalContainment[[BooleanInterval, Boolean,  $\wedge$ ]],
    BinaryIntervalContainment[[BooleanInterval, Boolean,  $\vee$ ]],
    BinaryIntervalContainment[[BooleanInterval, Boolean,  $\underline{\vee}$ ]],
    BinaryIntervalContainment[[BooleanInterval, Boolean,  $\equiv$ ]],
    BinaryIntervalContainment[[BooleanInterval, Boolean,  $=$ ]],
    BinaryIntervalContainment[[BooleanInterval, Boolean,  $\leftrightarrow$ ]],
    BinaryIntervalContainment[[BooleanInterval, Boolean,  $\overline{\wedge}$ ]],
    BinaryIntervalContainment[[BooleanInterval, Boolean,  $\overline{\vee}$ ]],
    BinaryIntervalContainment[[BooleanInterval, Boolean,  $\rightarrow$ ]],
    UnaryIntervalContainment[[BooleanInterval, Boolean,  $\neg$ ]],
    Generator[[Boolean]] }

  comprises { ... }
  coercion (x: Boolean)
  opr  $\wedge$ (self, other: BooleanInterval): BooleanInterval
  opr  $\vee$ (self, other: BooleanInterval): BooleanInterval
  opr  $\neg$ (self): BooleanInterval
  opr  $\underline{\vee}$ (self, other: BooleanInterval): BooleanInterval
  opr  $\oplus$ (self, other: BooleanInterval): BooleanInterval
  opr  $\equiv$ (self, other: BooleanInterval): BooleanInterval
  opr  $=$ (self, other: BooleanInterval): BooleanInterval
  opr  $\leftrightarrow$ (self, other: BooleanInterval): BooleanInterval
  opr  $\rightarrow$ (self, other: BooleanInterval): BooleanInterval
  opr  $\overline{\wedge}$ (self, other: BooleanInterval): BooleanInterval
  opr  $\overline{\vee}$ (self, other: BooleanInterval): BooleanInterval
  opr  $\in$ (other: Boolean, self): Boolean
  opr  $\cap$ (self, other: BooleanInterval): BooleanInterval
  opr  $\cup$ (self, other: BooleanInterval): BooleanInterval
  opr SET_COMPLEMENT(self): BooleanInterval
  opr SYMDIFF(self, other: BooleanInterval): BooleanInterval
  opr  $\setminus$ (self, other: BooleanInterval): BooleanInterval
  possibly(self): Boolean
  necessarily(self): Boolean
  certainly(self): Boolean
  opr  $\equiv$ (self, other: BooleanInterval): Boolean
  getter hashCode(): Z64
  toString(): String
  property true  $\in$  True  $\wedge$  false  $\notin$  True
  property true  $\notin$  False  $\wedge$  false  $\in$  False
  property true  $\in$  Uncertain  $\wedge$  false  $\in$  Uncertain
  property true  $\notin$  Impossible  $\wedge$  false  $\notin$  Impossible
  property  $\forall(a)$  necessarily(a)  $\equiv$   $\neg$ possibly( $\neg$ a)
  property  $\forall(a)$  possibly(a)  $\leftrightarrow$  true  $\in$  a
  property  $\forall(a)$  certainly(a)  $\leftrightarrow$  (true  $\in$  a  $\wedge$  false  $\notin$  a)
  property  $\forall(a, b)$  (a  $\overline{\wedge}$  b)  $\leftrightarrow$   $\neg$ (a  $\wedge$  b)
  property  $\forall(a, b)$  (a  $\overline{\vee}$  b)  $\leftrightarrow$   $\neg$ (a  $\vee$  b)
end
True: BooleanInterval
False: BooleanInterval
Uncertain: BooleanInterval

```

Impossible: BooleanInterval
`test testData[] = { True, False, Uncertain, Impossible }`

24.2.1 coercion (x : Boolean)

A boolean value can always serve as a BooleanInterval value. The value *true* is coerced to True; the value *false* is coerced to False.

24.2.2 `opr ^ (self, other: BooleanInterval): BooleanInterval`

The logical AND operator \wedge (AND) returns Impossible if either argument is Impossible; otherwise it returns False if either argument is False; otherwise it returns Uncertain if either argument is Uncertain; otherwise it returns True. It obeys the interval containment rule. The \wedge operator may be described by this table:

\wedge	Uncertain	True	False	Impossible
Uncertain	Uncertain	Uncertain	False	Impossible
True	Uncertain	True	False	Impossible
False	False	False	False	Impossible
Impossible	Impossible	Impossible	Impossible	Impossible

24.2.3 `opr v (self, other: BooleanInterval): BooleanInterval`

The logical OR operator \vee (OR) returns Impossible if either argument is Impossible; otherwise it returns True if either argument is True; otherwise it returns Uncertain if either argument is Uncertain; otherwise it returns False. It obeys the interval containment rule. The \vee operator may be described by this table:

\vee	Uncertain	True	False	Impossible
Uncertain	Uncertain	True	Uncertain	Impossible
True	True	True	True	Impossible
False	Uncertain	True	False	Impossible
Impossible	Impossible	Impossible	Impossible	Impossible

24.2.4 `opr ~ (self): BooleanInterval`

The logical NOT operator \neg (NOT) returns Impossible if its argument is Impossible, Uncertain if its argument is Uncertain, False if its argument is True, and True if its argument is False. It obeys the interval containment rule.

24.2.5 opr $\underline{\vee}$ (self, other: BooleanInterval): BooleanInterval

24.2.6 opr \oplus (self, other: BooleanInterval): BooleanInterval

The logical exclusive OR operator $\underline{\vee}$ (XOR) returns Impossible if either argument is Impossible; otherwise it returns Uncertain if either argument is Uncertain; otherwise it returns False if the arguments are strictly equivalent; otherwise it returns True. It obeys the interval containment rule.

The operator \oplus (OPLUS) does the same thing as $\underline{\vee}$. The $\underline{\vee}$ or \oplus operator may be described by this table:

$\underline{\vee}$ or \oplus	Uncertain	True	False	Impossible
Uncertain	Uncertain	Uncertain	Uncertain	Impossible
True	Uncertain	False	True	Impossible
False	Uncertain	True	False	Impossible
Impossible	Impossible	Impossible	Impossible	Impossible

24.2.7 opr \equiv (self, other: BooleanInterval): BooleanInterval

24.2.8 opr $=$ (self, other: BooleanInterval): BooleanInterval

24.2.9 opr \leftrightarrow (self, other: BooleanInterval): BooleanInterval

The logical equivalence, or exclusive NOR, operator \equiv (EQV) returns Impossible if either argument is Impossible; otherwise it returns Uncertain if either argument is Uncertain; otherwise it returns True if the arguments are strictly equivalent; otherwise it returns False. It obeys the interval containment rule. (Thus its behavior on boolean interval values is *not* the same as that of the strict equivalence operator \equiv .)

The equality operator $=$ and the if-and-only-if operator \leftrightarrow (IFF) do the same thing as \equiv . The \equiv or $=$ or \leftrightarrow operator may be described by this table:

\equiv or $=$ or \leftrightarrow	Uncertain	True	False	Impossible
Uncertain	Uncertain	Uncertain	Uncertain	Impossible
True	Uncertain	True	False	Impossible
False	Uncertain	False	True	Impossible
Impossible	Impossible	Impossible	Impossible	Impossible

24.2.10 opr \rightarrow (self, other: BooleanInterval): BooleanInterval

The logical implication operator \rightarrow (IMPLIES) returns Impossible if either argument is Impossible; otherwise it returns True if the first argument is False or the second argument is True; otherwise it returns Uncertain if either argument is Uncertain; otherwise it returns False. It obeys the interval containment rule. The \rightarrow operator may be described by this table:

\rightarrow	Uncertain	True	False	Impossible
Uncertain	Uncertain	True	Uncertain	Impossible
True	Uncertain	True	False	Impossible
False	True	True	True	Impossible
Impossible	Impossible	Impossible	Impossible	Impossible

24.2.11 opr $\bar{\wedge}$ (self, other: BooleanInterval): BooleanInterval

The logical NAND (NOT AND) operator $\bar{\wedge}$ (NAND) returns Impossible if either argument is Impossible; otherwise it returns True if either argument is False; otherwise it returns Uncertain if either argument is Uncertain; otherwise it returns False. It obeys the interval containment rule. The $\bar{\wedge}$ operator may be described by this table:

$\bar{\wedge}$	Uncertain	True	False	Impossible
Uncertain	Uncertain	Uncertain	True	Impossible
True	Uncertain	False	True	Impossible
False	True	True	True	Impossible
Impossible	Impossible	Impossible	Impossible	Impossible

24.2.12 opr $\bar{\vee}$ (self, other: BooleanInterval): BooleanInterval

The logical NOR (NOT OR) operator $\bar{\vee}$ (NOR) returns Impossible if either argument is Impossible; otherwise it returns False if either argument is True; otherwise it returns Uncertain if either argument is Uncertain; otherwise it returns True. It obeys the interval containment rule. The $\bar{\vee}$ operator may be described by this table:

$\bar{\vee}$	Uncertain	True	False	Impossible
Uncertain	Uncertain	False	Uncertain	Impossible
True	False	False	False	Impossible
False	Uncertain	False	True	Impossible
Impossible	Impossible	Impossible	Impossible	Impossible

24.2.13 opr \in (other: Boolean, self): Boolean

The operator \in (IN) returns *true* if its first argument, a boolean value, is contained in its second argument, a boolean interval regarded as a set; otherwise it returns *false*. The \in operator may be described by this table:

\in	Uncertain	True	False	Impossible
<i>true</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>false</i>
<i>false</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>false</i>

24.2.14 opr \cap (self, other: BooleanInterval): BooleanInterval

The intersection operator \cap (INTERSECTION or CAP) returns Impossible if either argument is Impossible; otherwise, if either argument is Uncertain, it returns the other argument; otherwise, if the arguments are the same value (strictly equivalent), it returns that value; otherwise it returns Impossible. The \cap operator may be described by this table:

\cap	Uncertain	True	False	Impossible
Uncertain	Uncertain	True	False	Impossible
True	True	True	Impossible	Impossible
False	False	Impossible	False	Impossible
Impossible	Impossible	Impossible	Impossible	Impossible

24.2.15 `opr ∪(self, other: BooleanInterval): BooleanInterval`

The union operator `∪` (`UNION` or `CUP`) returns `Uncertain` if either argument is `Uncertain`; otherwise, if either argument is `Impossible`, it returns the other argument; otherwise, if the arguments are the same value (strictly equivalent), it returns that value; otherwise it returns `Uncertain`. The `∪` operator may be described by this table:

<code>∪</code>	<code>Uncertain</code>	<code>True</code>	<code>False</code>	<code>Impossible</code>
<code>Uncertain</code>	<code>Uncertain</code>	<code>Uncertain</code>	<code>Uncertain</code>	<code>Uncertain</code>
<code>True</code>	<code>Uncertain</code>	<code>True</code>	<code>Uncertain</code>	<code>True</code>
<code>False</code>	<code>Uncertain</code>	<code>Uncertain</code>	<code>False</code>	<code>False</code>
<code>Impossible</code>	<code>Uncertain</code>	<code>True</code>	<code>False</code>	<code>Impossible</code>

24.2.16 `opr SET_COMPLEMENT(self): BooleanInterval`

The set complement operator `SET_COMPLEMENT` returns `Uncertain` if its argument is `Impossible`, `Impossible` if its argument is `Uncertain`, `False` if its argument is `True`, and `True` if its argument is `False`.

24.2.17 `opr SYMDIFF(self, other: BooleanInterval): BooleanInterval`

The symmetric difference operator `SYMDIFF` produces a result that contains a given boolean value if and only if exactly one of the arguments contains that boolean value. The `SYMDIFF` operator may be described by this table:

<code>SYMDIFF</code>	<code>Uncertain</code>	<code>True</code>	<code>False</code>	<code>Impossible</code>
<code>Uncertain</code>	<code>Impossible</code>	<code>False</code>	<code>True</code>	<code>Uncertain</code>
<code>True</code>	<code>False</code>	<code>Impossible</code>	<code>Uncertain</code>	<code>True</code>
<code>False</code>	<code>True</code>	<code>Uncertain</code>	<code>Impossible</code>	<code>False</code>
<code>Impossible</code>	<code>Uncertain</code>	<code>True</code>	<code>False</code>	<code>Impossible</code>

24.2.18 `opr \ (self, other: BooleanInterval): BooleanInterval`

The set difference operator `\` (`SETMINUS`) produces a result that contains a given boolean value if and only if the first argument contains that boolean value but the second argument does not. The `\` operator may be described by this table:

<code>\</code>	<code>Uncertain</code>	<code>True</code>	<code>False</code>	<code>Impossible</code>
<code>Uncertain</code>	<code>Impossible</code>	<code>False</code>	<code>True</code>	<code>Uncertain</code>
<code>True</code>	<code>Impossible</code>	<code>Impossible</code>	<code>True</code>	<code>True</code>
<code>False</code>	<code>Impossible</code>	<code>False</code>	<code>Impossible</code>	<code>False</code>
<code>Impossible</code>	<code>Impossible</code>	<code>Impossible</code>	<code>Impossible</code>	<code>Impossible</code>

- 24.2.19** *possibly*(self): Boolean
24.2.20 *necessarily*(self): Boolean
24.2.21 *certainly*(self): Boolean

The predicate *possibly* returns *true* if and only if *true* is a member of this boolean interval.

The predicate *necessarily* returns *true* if and only if *false* is not a member of this boolean interval (thus “necessarily” is a concise way of saying “not possibly not”).

The predicate *certainly* returns *true* if and only if this boolean interval is True, that is, it contains *true* but not *false* (thus “certainly” is a concise way of saying “both possibly and necessarily”).

The fourteen nontrivial functions from a value *x* of type BooleanInterval to type Boolean may thus be expressed as follows:

$$\begin{aligned}
 & \textit{necessarily}(x) \wedge \textit{necessarily}(\neg x) \\
 & \quad \textit{certainly}(\neg x) \\
 & \quad \textit{necessarily}(\neg x) \\
 & \quad \quad \textit{certainly}(x) \\
 & \quad \quad \textit{necessarily}(x) \\
 & \textit{possibly}(x) \equiv \textit{necessarily}(x) \\
 & \textit{necessarily}(x) \vee \textit{necessarily}(\neg x) \\
 & \quad \textit{possibly}(x) \wedge \textit{possibly}(\neg x) \\
 & \quad \textit{possibly}(x) \vee \textit{necessarily}(x) \\
 & \quad \quad \textit{possibly}(\neg x) \\
 & \quad \quad \neg \textit{certainly}(x) \\
 & \quad \quad \textit{possibly}(x) \\
 & \quad \quad \neg \textit{certainly}(\neg x) \\
 & \textit{possibly}(x) \vee \textit{possibly}(\neg x)
 \end{aligned}$$

There are other ways to express some of them; for example, $\textit{necessarily}(x) \wedge \textit{necessarily}(\neg x)$ is the same as $x \equiv \text{Impossible}$, and $\textit{possibly}(x) \wedge \textit{possibly}(\neg x)$ is the same as $x \equiv \text{Uncertain}$.

- 24.2.22** *opr* \equiv (self, other: BooleanInterval): Boolean

Two boolean intervals are strictly equivalent if and only if they are the same boolean interval.

- 24.2.23** *getter hashCode*(): Z64

- 24.2.24** *toString*(): String

The *toString* method returns either “True” or “False” or “Uncertain” or “Impossible” as appropriate.

24.3 Top-level BooleanInterval Values

24.3.1 True: BooleanInterval

24.3.2 False: BooleanInterval

24.3.3 Uncertain: BooleanInterval

24.3.4 Impossible: BooleanInterval

The immutable variables True, False, Uncertain, and Impossible have as their values the four boolean intervals. They are top-level variables declared in the Fortress standard libraries.

Chapter 25

Numbers

25.1 Rational Numbers

The trait \mathbb{Q} (`QQ`) encompasses all finite rational numbers, the results of dividing any integer by any nonzero integer. The trait \mathbb{Q}^* (`QQ_star`) is \mathbb{Q} with two extra elements, $+\infty$ and $-\infty$. The trait $\mathbb{Q}^\#$ (`QQ_splat`) is \mathbb{Q}^* with one additional element, the indefinite rational (written $0/0$), which is used as the result of dividing zero by zero or of adding $-\infty$ to $+\infty$.

Often it is desirable to indicate that a variable ranges over only a subset of the rationals, such as only positive values or only nonnegative values or only nonzero values. Unfortunately, traditional notations such as \mathbb{Q}^+ are not used consistently in the literature; one author may use \mathbb{Q}^+ to mean the set of strictly positive rationals and another may use it to mean the set of nonnegative rationals. Fortress therefore uses a notation that is novel but unambiguous:

\mathbb{Q} (`QQ`) is the set of rationals (it is a subtype of \mathbb{R} and \mathbb{Q}^*).

$\mathbb{Q}_{<}$ (`QQ_LT`) is the set of strictly negative rationals (it is a subtype of $\mathbb{R}_{<}$, $\mathbb{Q}_{<}^*$, \mathbb{Q} , \mathbb{Q}_{\leq} , and \mathbb{Q}_{\neq}).

\mathbb{Q}_{\leq} (`QQ_LE`) is the set of nonpositive rationals, that is, $\mathbb{Q}_{<} \cup \{0\}$ (it is a subtype of \mathbb{R}_{\leq} , \mathbb{Q}_{\leq}^* , and \mathbb{Q}).

\mathbb{Q}_{\geq} (`QQ_GE`) is the set of nonnegative rationals, that is, $\mathbb{Q}_{>} \cup \{0\}$ (it is a subtype of \mathbb{R}_{\geq} , \mathbb{Q}_{\geq}^* , and \mathbb{Q}).

$\mathbb{Q}_{>}$ (`QQ_GT`) is the set of strictly positive rationals (it is a subtype of $\mathbb{R}_{>}$, $\mathbb{Q}_{>}^*$, \mathbb{Q} , \mathbb{Q}_{\geq} , and \mathbb{Q}_{\neq}).

\mathbb{Q}_{\neq} (`QQ_NE`) is the set of strictly nonzero rationals (that is, $\mathbb{Q}_{<} \cup \mathbb{Q}_{>}$) (it is a subtype of \mathbb{R}_{\neq} , \mathbb{Q}_{\neq}^* , and \mathbb{Q}).

\mathbb{Q}^* (`QQ_star`) is \mathbb{Q} with extra elements $+\infty$ and $-\infty$ (it is a subtype of \mathbb{R}^* and $\mathbb{Q}^\#$).

$\mathbb{Q}_{<}^*$ (`QQ_star_LT`) is $\mathbb{Q}_{<}$ with extra element $-\infty$ (it is a subtype of $\mathbb{R}_{<}^*$, $\mathbb{Q}_{<}^\#$, \mathbb{Q}^* , \mathbb{Q}_{\leq}^* , and \mathbb{Q}_{\neq}^*).

\mathbb{Q}_{\leq}^* (`QQ_star_LE`) is \mathbb{Q}_{\leq} with extra element $-\infty$ (it is a subtype of \mathbb{R}_{\leq}^* , $\mathbb{Q}_{\leq}^\#$, and \mathbb{Q}^*).

\mathbb{Q}_{\geq}^* (`QQ_star_GE`) is \mathbb{Q}_{\geq} with extra element $+\infty$ (it is a subtype of \mathbb{R}_{\geq}^* , $\mathbb{Q}_{\geq}^\#$, and \mathbb{Q}^*).

$\mathbb{Q}_{>}^*$ (`QQ_star_GT`) is $\mathbb{Q}_{>}$ with extra element $+\infty$ (it is a subtype of $\mathbb{R}_{>}^*$, $\mathbb{Q}_{>}^\#$, \mathbb{Q}^* , \mathbb{Q}_{\geq}^* , and \mathbb{Q}_{\neq}^*).

\mathbb{Q}_{\neq}^* (`QQ_star_NE`) is \mathbb{Q}_{\neq} with extra elements $+\infty$ and $-\infty$ (it is a subtype of \mathbb{R}_{\neq}^* , $\mathbb{Q}_{\neq}^\#$, and \mathbb{Q}^*).

$\mathbb{Q}^\#$ (`QQ_splat`) is \mathbb{Q}^* with extra element $0/0$ (it is a subtype of $\mathbb{R}^\#$).

$\mathbb{Q}_{<}^\#$ (`QQ_splat_LT`) is $\mathbb{Q}_{<}^*$ with extra element $0/0$ (it is a subtype of $\mathbb{R}_{<}^\#$, $\mathbb{Q}^\#$, $\mathbb{Q}_{\leq}^\#$, and $\mathbb{Q}_{\neq}^\#$).

$\mathbb{Q}_{\leq}^\#$ (`QQ_splat_LE`) is \mathbb{Q}_{\leq}^* with extra element $0/0$ (it is a subtype of $\mathbb{R}_{\leq}^\#$ and $\mathbb{Q}^\#$).

$\mathbb{Q}_{\geq}^\#$ (`QQ_splat_GE`) is \mathbb{Q}_{\geq}^* with extra element $0/0$ (it is a subtype of $\mathbb{R}_{\geq}^\#$ and $\mathbb{Q}^\#$).

$\mathbb{Q}_{>}^\#$ (`QQ_splat_GT`) is $\mathbb{Q}_{>}^*$ with extra element $0/0$ (it is a subtype of $\mathbb{R}_{>}^\#$, $\mathbb{Q}^\#$, $\mathbb{Q}_{\geq}^\#$, and $\mathbb{Q}_{\neq}^\#$).

$\mathbb{Q}_{\neq}^\#$ (`QQ_splat_NE`) is \mathbb{Q}_{\neq}^* with extra element $0/0$ (it is a subtype of $\mathbb{R}_{\neq}^\#$ and $\mathbb{Q}^\#$).

The Fortress type system tracks these types closely through various arithmetic operations; for example, adding two values of type $\mathbb{Q}_{>}$ produces a result of type $\mathbb{Q}_{>}$, and adding a value of type \mathbb{Q}_{\geq}^* and a value of type \mathbb{Q}_{\geq} produces a value of type \mathbb{Q}_{\geq}^* .

Here we present only the trait \mathbb{Q} and its methods. The other rational types have exactly the same methods and differ only in the details of the types of method arguments and results and exactly what traits are extended by each rational type. For example, \mathbb{Q} is a field and is totally ordered, \mathbb{Q}^* is totally ordered but is not a field, and $\mathbb{Q}^\#$ is neither totally ordered nor a field. For the exact details of how all this is implemented, see Section 38.1.

```

trait Q
  extends { R, Q*,
           Field[Q, Q $\neq$ , +, -,  $\cdot$ , /],
           Field[Q, Q $\neq$ , +, -,  $\times$ , /],
           Field[Q, Q $\neq$ , +, -, juxtaposition, /],
           TotalOrderOperators[Q, <,  $\leq$ ,  $\geq$ , >, CMP],
           PartialOrderAndLattice[Q,  $\leq$ , MIN, MAX] }

  coercion (x: Identity[+])
  coercion (x: Identity[ $\cdot$ ])
  coercion (x: Identity[ $\times$ ])
  coercion (x: Identity[juxtaposition])
  coercion (x: Zero[ $\cdot$ ])
  coercion (x: Zero[ $\times$ ])
  coercion (x: Zero[juxtaposition])
  opr juxtaposition (self, other: Q): Q
  opr +(self): Q
  opr +(self, other: Q): Q
  opr -(self): Q
  opr -(self, other: Q): Q
  opr  $\cdot$ (self, other: Q): Q
  opr  $\times$ (self, other: Q): Q
  opr /(self): Q*
  opr /(self, other: Q): Q#
  opr  $\wedge$ (self, power: Z): Q#
  opr <(self, other: Q): Boolean
  opr  $\leq$ (self, other: Q): Boolean
  opr =(self, other: Q): Boolean
  opr  $\geq$ (self, other: Q): Boolean
  opr >(self, other: Q): Boolean
  opr CMP(self, other: Q*): TotalComparison
  opr CMP(self, other: Q#): Comparison
  opr MAX(self, other: Q): Q
  opr MIN(self, other: Q): Q
  opr MAXNUM(self, other: Q): Q
  opr MINNUM(self, other: Q): Q
  opr |self| : Q $\geq$ 
  signum(self): Z
  numerator(self): Z
  denominator(self): Z
  floor(self): Z
  opr  $\lfloor$ self $\rfloor$ : Z
  ceiling(self): Z
  opr  $\lceil$ self $\rceil$ : Z
  round(self): Z
  truncate(self): Z
  opr  $\llbracket$ self $\rrbracket$ : N
  opr  $\llbracket$ self $\rrbracket$ : N

```

```

opr  $\|self\|$ : $\mathbb{N}$ 
opr  $\|self\|$ : $\mathbb{N}$ 
realpart(self): $\mathbb{Q}$ 
imagpart(self): $\mathbb{Q}$ 
check(self): $\mathbb{Q}$  throws CastException
check*(self): $\mathbb{Q}^*$  throws CastException
check<(self): $\mathbb{Q}_{<}$  throws CastException
check≤(self): $\mathbb{Q}_{\leq}$  throws CastException
check≥(self): $\mathbb{Q}_{\geq}$  throws CastException
check>(self): $\mathbb{Q}_{>}$  throws CastException
check≠(self): $\mathbb{Q}_{\neq}$  throws CastException
check* $\leq$ (self): $\mathbb{Q}^*_{\leq}$  throws CastException
check* $\geq$ (self): $\mathbb{Q}^*_{\geq}$  throws CastException
check* $\leq$  $\leq$ (self): $\mathbb{Q}^*_{\leq\leq}$  throws CastException
check* $\geq$  $\geq$ (self): $\mathbb{Q}^*_{\geq\geq}$  throws CastException
check* $\leq$  $\geq$ (self): $\mathbb{Q}^*_{\leq\geq}$  throws CastException
check* $\geq$  $\leq$ (self): $\mathbb{Q}^*_{\geq\leq}$  throws CastException
check* $\neq$ (self): $\mathbb{Q}^*_{\neq}$  throws CastException
check $\neq$  $\leq$ (self): $\mathbb{Q}_{\neq\leq}$  throws CastException
check $\neq$  $\geq$ (self): $\mathbb{Q}_{\neq\geq}$  throws CastException
check $\leq$  $\neq$ (self): $\mathbb{Q}_{\leq\neq}$  throws CastException
check $\geq$  $\neq$ (self): $\mathbb{Q}_{\geq\neq}$  throws CastException
end

```

- 25.1.1** coercion (x : Identity[+])
- 25.1.2** coercion (x : Identity[.])
- 25.1.3** coercion (x : Identity[\times])
- 25.1.4** coercion (x : Identity[juxtaposition])
- 25.1.5** coercion (x : Zero[.])
- 25.1.6** coercion (x : Zero[\times])
- 25.1.7** coercion (x : Zero[juxtaposition])

The identity for + is 0.

The identity for juxtaposition or \cdot or \times is 1.

The zero for juxtaposition or \cdot or \times is 0.

- 25.1.8** opr juxtaposition (self, other: \mathbb{Q}): \mathbb{Q}

Juxtaposition of rational expressions is equivalent to using the multiplication operator \cdot .

- 25.1.9** opr +(self): \mathbb{Q}

The unary addition operator + simply returns its argument.

25.1.10 `opr +(self, other: \mathbb{Q}): \mathbb{Q}`

The binary addition operator `+` returns the sum of its arguments.

For types \mathbb{Q}^* and $\mathbb{Q}^\#$, the sum of an infinity and either a finite rational or another infinity of the same sign is equal to the given infinity, but the sum of infinities of differing sign is $0/0$, and the sum of $0/0$ and any rational value is $0/0$.

25.1.11 `opr -(self): \mathbb{Q}`

The unary negation operator `-` returns the negative of its argument.

For types \mathbb{Q}^* and $\mathbb{Q}^\#$, the negative of $+\infty$ is $-\infty$, the negative of $-\infty$ is $+\infty$, and the negative of $0/0$ is $0/0$.

25.1.12 `opr -(self, other: \mathbb{Q}): \mathbb{Q}`

The binary subtraction operator `-` returns the difference of its arguments, which is equal to the sum of (a) the first argument and (b) the negation of the second argument.

25.1.13 `opr *(self, other: \mathbb{Q}): \mathbb{Q}`

25.1.14 `opr *(self, other: \mathbb{Q}): \mathbb{Q}`

The multiplication operator `*` (`DOT`) returns the product of its arguments. The multiplication operator `*` (`TIMES`) does exactly the same thing.

For types \mathbb{Q}^* and $\mathbb{Q}^\#$, the product of $0/0$ and any rational value is $0/0$, and the product of zero and an infinity (regardless of sign) is $0/0$; the product of an infinity and any rational value other than zero and $0/0$ is an infinity whose sign is positive if and only if the two arguments have the same sign.

25.1.15 `opr /(self): \mathbb{Q}^*`

The unary reciprocal operator `/` returns the reciprocal of its argument. The reciprocal of zero is $+\infty$ (and therefore the result type of `/` when given an argument of type \mathbb{Q} is necessarily \mathbb{Q}^*).

For types \mathbb{Q}^* and $\mathbb{Q}^\#$, the reciprocal of either $+\infty$ or $-\infty$ is zero, and the reciprocal of $0/0$ is $0/0$.

25.1.16 `opr /(self, other: \mathbb{Q}): \mathbb{Q}^*`

The binary division operator `/` returns the quotient of its arguments, which is equal to the product of (a) the first argument and (b) the reciprocal of the second argument.

25.1.17 `opr ^ (self, power: ℤ): ℚ#`

Exponentiation of a rational number to an integer power produces a rational result. If the *power* is 0, then the result is always 1, even if the rational number base is 0 (this definition is somewhat arbitrary but is computationally useful).

```
property ∀(x, y: ℤ) xy = 1/(x-y)
property ∀(x, y: ℤ) xy = x(⌊y/2⌋)x(⌈y/2⌉)
```

25.1.18 `opr < (self, other: ℚ): Boolean`

25.1.19 `opr ≤ (self, other: ℚ): Boolean`

25.1.20 `opr = (self, other: ℚ): Boolean`

25.1.21 `opr ≥ (self, other: ℚ): Boolean`

25.1.22 `opr > (self, other: ℚ): Boolean`

The comparison operators `<`, `≤`, `=`, `≥`, and `>` allow any rational value to be compared numerically to any other rational value.

For types \mathbb{Q}^* and $\mathbb{Q}^\#$, the rational values are totally ordered except for $0/0$, which is unordered with respect to all other rational values; moreover, for compatibility with floating-point arithmetic, $0/0$ is unordered with respect to itself, and therefore these five comparison operators always return *false* if either argument is $0/0$. The value $-\infty$ is less than any finite rational value, and $+\infty$ is greater than any finite rational value.

For `≠` see Section 26.1.4.

25.1.23 `opr CMP (self, other: ℚ): TotalComparison`

25.1.24 `opr CMP (self, other: ℚ#): Comparison`

The `CMP` operator compares the arguments and returns one of the four values `LessThan`, `EqualTo`, `GreaterThan`, and `Unordered`. If the argument types are such that the result cannot be `Unordered`, then the result has type `TotalComparison` rather than simply `Comparison`.

25.1.25 `opr MAX (self, other: ℚ): ℚ`

25.1.26 `opr MIN (self, other: ℚ): ℚ`

25.1.27 `opr MAXNUM (self, other: ℚ): ℚ`

25.1.28 `opr MINNUM (self, other: ℚ): ℚ`

The operators `MAX` and `MAXNUM` return whichever argument is larger in the total order defined by `<`, `≤`, `=`, `≥`, `>`, and `CMP`, and the operators `MIN` and `MINNUM` return whichever argument is smaller. (For all four, if the arguments are equal, then the result equals that same value.)

For type $\mathbb{Q}^\#$, `MAXNUM` and `MINNUM` differ from `MAX` and `MIN` in their treatment of $0/0$: if one argument is $0/0$ and the other is not, then `MAX` or `MIN` returns $0/0$ but `MAXNUM` or `MINNUM` returns the argument that is not $0/0$.

25.1.29 `opr |self| : \mathbb{Q}_{\geq}`

The absolute value operator `|...|` returns the negative of this rational number if the argument is less than zero, and otherwise returns the argument.

For type $\mathbb{Q}^{\#}$, the absolute value of $0/0$ is $0/0$.

25.1.30 `signum(self) : \mathbb{Z}`

The method `signum` returns -1 if this rational number is less than zero, 0 if this rational number is zero, and 1 if this rational number is greater than zero.

For type $\mathbb{Q}^{\#}$, the signum of $0/0$ is $0/0$.

25.1.31 `numerator(self) : \mathbb{Z}`

25.1.32 `denominator(self) : \mathbb{Z}`

The method `numerator` returns the numerator of this rational number, and the method `denominator` returns the denominator of this rational number, when this rational number is represented in lowest terms (such that the greatest common divisor of numerator and denominator is 1).

For types \mathbb{Q}^* and $\mathbb{Q}^{\#}$, the numerator of $+\infty$ is 1 , the numerator of $-\infty$ is -1 , and the numerator of $0/0$ is 0 ; the denominator of $+\infty$, $-\infty$, or $0/0$ is 0 .

- 25.1.33** `floor(self): ℤ`
- 25.1.34** `opr ⌊self⌋: ℤ`
- 25.1.35** `ceiling(self): ℤ`
- 25.1.36** `opr ⌈self⌉: ℤ`
- 25.1.37** `round(self): ℤ`
- 25.1.38** `truncate(self): ℤ`

The method `floor`, likewise the enclosing operator $\lfloor \dots \rfloor$, returns the largest integer that is not greater than this rational number.

The method `ceiling`, likewise the enclosing operator $\lceil \dots \rceil$, returns the smallest integer that is not less than this rational number.

The method `round` returns the integer that is closest to this rational number, but if this rational number is exactly halfway between two consecutive integers, then `round` returns whichever of the two integers is even.

The method `truncate` returns the ceiling of this rational number if it is negative, and otherwise returns the floor of this rational number. (This has the effect of taking the floor of the magnitude, also called “rounding toward zero.”)

For types \mathbb{Q}^* and $\mathbb{Q}^\#$, all of these methods simply return the argument if it is $+\infty$, $-\infty$, or $0/0$.

- `opr ⌊self⌋: ℕ`
- `opr ⌈self⌉: ℕ`
- `opr ⌊⌊self⌋⌋: ℕ`
- `opr ⌈⌈self⌉⌉: ℕ`

The hyperfloor operation $\llbracket x \rrbracket$ computes $2^{\lfloor \log_2 x \rfloor}$ and returns the result as a natural number. If the argument is equal to 0, the result is 0. If the argument is negative, an `InvalidArgumentException` is thrown.

The hyperceiling operation $\lceil\lceil x \rceil\rceil$ computes $2^{\lceil \log_2 x \rceil}$ and returns the result as a natural number. If the argument is equal to 0, the result is 0. If the argument is negative, an `InvalidArgumentException` is thrown.

The hyperhyperfloor operation $\lllbracket x \rrracket$ computes $2^{\llbracket \log_2 x \rrbracket}$ and returns the result as a natural number. If the argument is equal to 0 or 1, the result is the same as the argument. If the argument is negative, an `InvalidArgumentException` is thrown.

The hyperhyperceiling operation $\lllceil x \rrlceil$ computes $2^{\lceil\lceil \log_2 x \rceil\rceil}$ and returns the result as a natural number. If the argument is equal to 0 or 1, the result is the same as the argument. If the argument is negative, an `InvalidArgumentException` is thrown.

- 25.1.39** `realpart(self): ℚ`

The method `realpart` for a rational number simply returns its argument.

- 25.1.40** `imagpart(self): ℚ`

The method `imagpart` for a rational number simply returns zero.

- 25.1.41 *check*(self): \mathbb{Q} throws `CastException`
- 25.1.42 *check**(self): \mathbb{Q}^* throws `CastException`
- 25.1.43 *check*_<(self): $\mathbb{Q}_{<}$ throws `CastException`
- 25.1.44 *check*_≤(self): \mathbb{Q}_{\leq} throws `CastException`
- 25.1.45 *check*_≥(self): \mathbb{Q}_{\geq} throws `CastException`
- 25.1.46 *check*_>(self): $\mathbb{Q}_{>}$ throws `CastException`
- 25.1.47 *check*_≠(self): \mathbb{Q}_{\neq} throws `CastException`
- 25.1.48 *check*_<^{*}(self): $\mathbb{Q}_{<}^*$ throws `CastException`
- 25.1.49 *check*_≤^{*}(self): \mathbb{Q}_{\leq}^* throws `CastException`
- 25.1.50 *check*_≥^{*}(self): \mathbb{Q}_{\geq}^* throws `CastException`
- 25.1.51 *check*_>^{*}(self): $\mathbb{Q}_{>}^*$ throws `CastException`
- 25.1.52 *check*_≠^{*}(self): \mathbb{Q}_{\neq}^* throws `CastException`
- 25.1.53 *check*_<[#](self): $\mathbb{Q}_{<}^{\#}$ throws `CastException`
- 25.1.54 *check*_≤[#](self): $\mathbb{Q}_{\leq}^{\#}$ throws `CastException`
- 25.1.55 *check*_≥[#](self): $\mathbb{Q}_{\geq}^{\#}$ throws `CastException`
- 25.1.56 *check*_>[#](self): $\mathbb{Q}_{>}^{\#}$ throws `CastException`
- 25.1.57 *check*_≠[#](self): $\mathbb{Q}_{\neq}^{\#}$ throws `CastException`

Each of these methods checks this rational number to see whether it belongs to the result type of the method. If, the number is returned; if not, a `CastException` is thrown.

25.2 Integers

The trait \mathbb{Z} (`ZZ`) encompasses all finite integers, the results of starting from 0 and repeatedly adding or subtracting 1 a finite number of times. The trait \mathbb{Z}^* (`ZZ_star`) is \mathbb{Z} with two extra elements, $+\infty$ and $-\infty$.

Often it is desirable to indicate that a variable ranges over only a subset of the integers, such as only positive values or only nonnegative values or only nonzero values. Fortress uses a system similar to that for rational numbers, but also accommodates the traditional use of \mathbb{N} to denote the natural numbers:

\mathbb{Z} (`ZZ`) is the set of integers (it is a subtype of \mathbb{Q} and \mathbb{Z}^*).

$\mathbb{Z}_{<}$ (`ZZ_LT`) is the set of strictly negative integers (it is a subtype of $\mathbb{Q}_{<}$, \mathbb{Z}^* , \mathbb{Z} , \mathbb{Z}_{\leq} , and \mathbb{Z}_{\neq}).

\mathbb{Z}_{\leq} (`ZZ_LE`) is the set of nonpositive integers, that is, $\mathbb{Z}_{<} \cup \{0\}$ (it is a subtype of \mathbb{Q}_{\leq} , \mathbb{Z}_{\leq}^* , and \mathbb{Z}).

\mathbb{Z}_{\geq} (`ZZ_GE`) is the set of nonnegative integers, that is, $\mathbb{Z}_{>} \cup \{0\}$ (it is a subtype of \mathbb{Q}_{\geq} , \mathbb{Z}_{\geq}^* , and \mathbb{Z}).

$\mathbb{Z}_{>}$ (`ZZ_GT`) is the set of strictly positive integers (it is a subtype of $\mathbb{Q}_{>}$, \mathbb{Z}^* , \mathbb{Z} , \mathbb{Z}_{\geq} , and \mathbb{Z}_{\neq}).

\mathbb{Z}_{\neq} (`ZZ_NE`) is the set of strictly nonzero integers (that is, $\mathbb{Z}_{<} \cup \mathbb{Z}_{>}$) (it is a subtype of \mathbb{Q}_{\neq} , \mathbb{Z}_{\neq}^* , and \mathbb{Z}).

\mathbb{Z}^* (`ZZ_star`) is \mathbb{Z} with extra elements $+\infty$ and $-\infty$ (it is a subtype of \mathbb{Q}^*).

$\mathbb{Z}_{<}^*$ (`ZZ_star_LT`) is $\mathbb{Z}_{<}$ with extra element $-\infty$ (it is a subtype of $\mathbb{Q}_{<}^*$, \mathbb{Z}^* , \mathbb{Z}_{\leq}^* , and \mathbb{Z}_{\neq}^*).

\mathbb{Z}_{\leq}^* (`ZZ_star_LE`) is \mathbb{Z}_{\leq} with extra element $-\infty$ (it is a subtype of \mathbb{Q}_{\leq}^* and \mathbb{Z}^*).

\mathbb{Z}_{\geq}^* (`ZZ_star_GE`) is \mathbb{Z}_{\geq} with extra element $+\infty$ (it is a subtype of \mathbb{Q}_{\geq}^* and \mathbb{Z}^*).

$\mathbb{Z}_{>}^*$ (`ZZ_star_GT`) is $\mathbb{Z}_{>}$ with extra element $+\infty$ (it is a subtype of $\mathbb{Q}_{>}^*$, \mathbb{Z}^* , \mathbb{Z}_{\geq}^* , and \mathbb{Z}_{\neq}^*).

\mathbb{Z}_{\neq}^* (`ZZ_star_NE`) is \mathbb{Z}_{\neq} with extra elements $+\infty$ and $-\infty$ (it is a subtype of \mathbb{Q}_{\neq}^* and \mathbb{Z}^*).

\mathbb{N} (`NN`) is a synonym for \mathbb{Z}_{\geq} .

\mathbb{N}^* (`NN_star`) is a synonym for \mathbb{Z}_{\geq}^* .

The Fortress type system tracks these types closely through various arithmetic operations; for example, adding two values of type $\mathbb{Z}_{>}$ produces a result of type $\mathbb{Z}_{>}$, and adding a value of type $\mathbb{Z}_{>}^*$ and a value of type \mathbb{Z}_{\geq} produces a value of type \mathbb{Z}_{\geq}^* .

Here we present only the trait \mathbb{Z} and its methods. The other integer types have exactly the same methods and differ only in the details of the types of method arguments and results and exactly what traits are extended by each integer type. For example, \mathbb{Z} is a commutative ring and is totally ordered, and \mathbb{Z}^* is totally ordered but is not a ring. *Future versions of this specification will include the exact details of how all this is implemented.*

```

trait  $\mathbb{Z}$ 
  extends {  $\mathbb{Q}$ ,  $\mathbb{Z}^*$ , IntegerLike[ $\mathbb{Z}$ ],
           CommutativeRing[ $\mathbb{Z}$ , +, -, juxtaposition],
           CommutativeRing[ $\mathbb{Z}$ , +, -, ·],
           CommutativeRing[ $\mathbb{Z}$ , +, -, ×],
           CommutativeRing[ $\mathbb{Z}$ , ⊕, ⊖, ⊠],
           CommutativeRing[ $\mathbb{Z}$ , ⊕, ⊖, ⊗],
           CommutativeRing[ $\mathbb{Z}$ , †, ‡, ×̂],
           TotalOrderOperators[ $\mathbb{Z}$ , <, ≤, ≥, >, CMP],
           PartialOrderAndLattice[ $\mathbb{Z}$ , ≤, MIN, MAX],
           BooleanAlgebra[ $\mathbb{Z}$ , ∧, ∨, ¬, ⊞] }

  coercion (x: Identity[+])
  coercion (x: Identity[⊕])
  coercion (x: Identity[†])
  coercion (x: Identity[juxtaposition])
  coercion (x: Identity[·])
  coercion (x: Identity[×])
  coercion (x: Identity[⊠])
  coercion (x: Identity[⊗])

```

```

coercion (x: Identity[×])
coercion (x: Zero[juxtaposition])
coercion (x: Zero[.])
coercion (x: Zero[×])
coercion (x: Zero[□])
coercion (x: Zero[⊗])
coercion (x: Zero[×])
opr juxtaposition (self, other: ℤ): ℤ
opr +(self): ℤ
opr ⊕(self): ℤ
opr ⋅(self): ℤ
opr +(self, other: ℤ): ℤ
opr ⊕(self, other: ℤ): ℤ
opr ⋅(self, other: ℤ): ℤ
opr -(self): ℤ
opr ⊖(self): ℤ
opr ⋅(self): ℤ
opr -(self, other: ℤ): ℤ
opr ⊖(self, other: ℤ): ℤ
opr ⋅(self, other: ℤ): ℤ
opr ⋅(self, other: ℤ): ℤ
opr ×(self, other: ℤ): ℤ
opr □(self, other: ℤ): ℤ
opr ⊗(self, other: ℤ): ℤ
opr ×(self, other: ℤ): ℤ
opr /(self, other: ℤ): ℚ#
opr ÷(self, other: ℤ): ℤ throws IntegerDivideByZeroException
opr REM(self, other: ℤ): ℤ throws IntegerDivideByZeroException
opr MOD(self, other: ℤ): ℤ throws IntegerDivideByZeroException
opr DIVREM(self, other: ℤ): (ℤ, ℤ) throws IntegerDivideByZeroException
opr DIVMOD(self, other: ℤ): (ℤ, ℤ) throws IntegerDivideByZeroException
opr |(self, other: ℤ): Boolean
opr _(self, power: ℕ): ℤ
opr GCD(self, other: ℤ): ℤ
opr LCM(self, other: ℤ): ℤ
opr (self)!: ℕ
opr CHOOSE(self, other: ℤ): ℕ
opr <(self, other: ℤ): Boolean
opr ≤(self, other: ℤ): Boolean
opr =(self, other: ℤ): Boolean
opr ≥(self, other: ℤ): Boolean
opr >(self, other: ℤ): Boolean
opr CMP(self, other: ℤ*): TotalComparison
opr MAX(self, other: ℤ): ℤ
opr MIN(self, other: ℤ): ℤ
opr MAXNUM(self, other: ℤ): ℤ
opr MINNUM(self, other: ℤ): ℤ
opr |self| : ℤ≥
signum(self): ℤ
numerator(self): ℤ
denominator(self): ℤ
floor(self): ℤ

```

```

opr |self|: ℤ
ceiling(self): ℤ
opr ⌈self⌉: ℤ
round(self): ℤ
truncate(self): ℤ
opr ⌊self⌋: ℕ
opr ⌈self⌉: ℕ
opr ⌊self⌋: ℕ
opr ⌈self⌉: ℕ
opr ⌊self⌋: ℕ
shift(self, k: IndexInt): ℤ
bit(self, k: IndexInt): Bit
opr ¬(self): ℤ
opr ∧(self, other: ℤ): ℤ
opr ∨(self, other: ℤ): ℤ
opr ⊕(self, other: ℤ): ℤ
countBits(self): IndexInt
countFactorsOfTwo(self): IndexInt
integerLength(self): IndexInt
lowBits(self, k: IndexInt): ℤ
even(self): Boolean
odd(self): Boolean
prime(self): Boolean
realpart(self): ℤ
imagpart(self): ℤ
check(self): ℤ throws CastException
check*(self): ℤ* throws CastException
check<(self): ℤ< throws CastException
check≤(self): ℤ≤ throws CastException
check≥(self): ℤ≥ throws CastException
check>(self): ℤ> throws CastException
check≠(self): ℤ≠ throws CastException
check*<(self): ℤ*< throws CastException
check*≤(self): ℤ*≤ throws CastException
check*≥(self): ℤ*≥ throws CastException
check*>(self): ℤ*> throws CastException
check*≠(self): ℤ*≠ throws CastException
end

```

25.2.1 coercion ($x: \text{Identity}[+]$)
25.2.2 coercion ($x: \text{Identity}[\boxplus]$)
25.2.3 coercion ($x: \text{Identity}[\dot{+}]$)
25.2.4 coercion ($x: \text{Identity}[\text{juxtaposition}]$)
25.2.5 coercion ($x: \text{Identity}[\cdot]$)
25.2.6 coercion ($x: \text{Identity}[\times]$)
25.2.7 coercion ($x: \text{Identity}[\square]$)
25.2.8 coercion ($x: \text{Identity}[\boxtimes]$)
25.2.9 coercion ($x: \text{Identity}[\dot{\times}]$)
25.2.10 coercion ($x: \text{Zero}[\text{juxtaposition}]$)
25.2.11 coercion ($x: \text{Zero}[\cdot]$)
25.2.12 coercion ($x: \text{Zero}[\times]$)
25.2.13 coercion ($x: \text{Zero}[\square]$)
25.2.14 coercion ($x: \text{Zero}[\boxtimes]$)
25.2.15 coercion ($x: \text{Zero}[\dot{\times}]$)

The identity for $+$ or \boxplus or $\dot{+}$ is 0.

The identity for juxtaposition or \cdot or \times or \square or \boxtimes or $\dot{\times}$ is 1.

The zero for juxtaposition or \cdot or \times or \square or \boxtimes or $\dot{\times}$ is 0.

25.2.16 opr juxtaposition (self, other: \mathbb{Z}): \mathbb{Z}

Juxtaposition of integer expressions is equivalent to using the multiplication operator \cdot .

25.2.17 opr +(self): \mathbb{Z}

25.2.18 opr \boxplus (self): \mathbb{Z}

25.2.19 opr $\dot{+}$ (self): \mathbb{Z}

The unary addition operator $+$ simply returns its argument. The operators \boxplus and $\dot{+}$ do exactly the same thing.

25.2.20 opr +(self, other: \mathbb{Z}): \mathbb{Z}

25.2.21 opr \boxplus (self, other: \mathbb{Z}): \mathbb{Z}

25.2.22 opr $\dot{+}$ (self, other: \mathbb{Z}): \mathbb{Z}

The binary addition operator $+$ returns the sum of its arguments. The wrapping and saturating addition operators \boxplus and $\dot{+}$ do exactly the same thing, because operations on type \mathbb{Z} never need to wrap or saturate.

For type \mathbb{Z}^* , the sum of an infinity and either a finite integer or another infinity of the same sign is equal to the given infinity, but attempting to sum infinities of differing sign throws an `IndeterminateIntegerException`.

25.2.23 `opr -(self): ℚ`
25.2.24 `opr ⊖(self): ℚ`
25.2.25 `opr ⋈(self): ℚ`

The unary negation operator `-` returns the negative of its argument. The operators `⊖` and `⋈` do exactly the same thing.

For types \mathbb{Z}^* and $\mathbb{Z}^\#$, the negative of $+\infty$ is $-\infty$, and the negative of $-\infty$ is $+\infty$.

25.2.26 `opr -(self, other: ℤ): ℤ`
25.2.27 `opr ⊖(self, other: ℤ): ℤ`
25.2.28 `opr ⋈(self, other: ℤ): ℤ`

The binary subtraction operator `-` returns the difference of its arguments, which is equal to the sum of (a) the first argument and (b) the negation of the second argument. The wrapping and saturating subtraction operators `⊖` and `⋈` do exactly the same thing, because operations on type `ZZ` never need to wrap or saturate.

25.2.29 `opr ⋅(self, other: ℤ): ℤ`
25.2.30 `opr ×(self, other: ℤ): ℤ`
25.2.31 `opr ⊠(self, other: ℤ): ℤ`
25.2.32 `opr ⊞(self, other: ℤ): ℤ`
25.2.33 `opr ⋈(self, other: ℤ): ℤ`

The multiplication operator `⋅` returns the product of its arguments. The multiplication operators `×`, `⊠`, `⊞`, and `⋈` do exactly the same thing.

For types \mathbb{Z}^* and $\mathbb{Z}^\#$, attempting to multiply zero and an infinity (regardless of sign) throws an `IndeterminateIntegerException`. The product of an infinity and any nonzero integer (including an infinity) is an infinity whose sign is positive if and only if the two arguments have the same sign.

25.2.34 `opr /(self, other: ℤ): ℚ#`

The binary division operator `/` returns the quotient of its arguments as a rational number. Unlike in other programming languages such as Fortran, the operator `/` does not perform truncating integer division; the operator `÷` may be used for that purpose.

- 25.2.35** `opr ÷(self, other: ℤ): ℤ` throws `IntegerDivideByZeroException`
- 25.2.36** `opr REM(self, other: ℤ): ℤ` throws `IntegerDivideByZeroException`
- 25.2.37** `opr MOD(self, other: ℤ): ℤ` throws `IntegerDivideByZeroException`
- 25.2.38** `opr DIVREM(self, other: ℤ): (ℤ, ℤ)` throws `IntegerDivideByZeroException`
- 25.2.39** `opr DIVMOD(self, other: ℤ): (ℤ, ℤ)` throws `IntegerDivideByZeroException`

The operator `÷` performs truncating integer division; if the quotient is not an exact integer, the result is rounded toward zero.

The operator `REM` returns the remainder that would be left over from a division using the operator `÷`.

The operator `MOD` returns the remainder that would be left over from an integer division that rounds inexact results towards negative infinity (“floor division”).

The operator `DIVREM` returns a tuple of two results, the quotient and remainder from a truncating integer division.

The operator `DIVMOD` returns a tuple of two results, the quotient and remainder from an integer floor division.

For all these operators, if `other` is zero then an `IntegerDivideByZeroException` is thrown.

Examples:

<code>+8 ÷ +3 = +2</code>	<code>+8 REM +3 = +2</code>	<code>+8 MOD +3 = +2</code>	<code>+8 DIVREM +3 = (+2, +2)</code>	<code>+8 DIVMOD +3 = (+2, +2)</code>
<code>-8 ÷ +3 = -2</code>	<code>-8 REM +3 = -2</code>	<code>-8 MOD +3 = +1</code>	<code>-8 DIVREM +3 = (-2, -2)</code>	<code>-8 DIVMOD +3 = (-3, +1)</code>
<code>+8 ÷ -3 = -2</code>	<code>+8 REM -3 = +2</code>	<code>+8 MOD -3 = -1</code>	<code>+8 DIVREM -3 = (-2, +2)</code>	<code>+8 DIVMOD -3 = (-3, -1)</code>
<code>-8 ÷ -3 = +2</code>	<code>-8 REM -3 = -2</code>	<code>-8 MOD -3 = -2</code>	<code>-8 DIVREM -3 = (+2, -2)</code>	<code>-8 DIVMOD -3 = (+2, -2)</code>

These examples illustrate the fact that truncating division and floor division behave differently when the two operands are of opposite sign and their quotient is not an exact integer.

- `property ∀(m, n) m REM n = m - n(m ÷ n)`
- `property ∀(m, n) m MOD n = m - n⌊m/n⌋`
- `property ∀(m, n) m DIVREM n = (m ÷ n, m REM n)`
- `property ∀(m, n) m DIVMOD n = (⌊m/n⌋, m MOD n)`
- `property ∀(m, n, k) (m + kn) MOD n = m MOD n`
- `property ∀(m, n) m REM n = m REM (-n)`
- `property ∀(m, n) (-m) REM n = -(m REM n)`

- 25.2.40** `opr |(self, other: ℤ): Boolean`

The operator `|` returns `true` if the left-hand operand evenly divides the right-hand operand, and otherwise returns `false`.

- `property ∀(m, n) m | n := (⌊m/n⌋ = m/n)`

- 25.2.41** `opr ^(self, power: ℕ): ℤ`

Exponentiation of an integer to a nonnegative integer power produces an integer result. If the `power` is 0, then the result is always 1, even if the base is 0 (this definition is somewhat arbitrary but is computationally useful).

- `property ∀(x, y: ℕ) xy = x(⌊y/2⌋)x(⌈y/2⌉)`

25.2.42 opr GCD(*self*, *other*: \mathbb{Z}): \mathbb{Z}
25.2.43 opr LCM(*self*, *other*: \mathbb{Z}): \mathbb{Z}

The operator GCD computes the greatest common divisor of its two arguments. The result is always nonnegative. If either argument is 0, the result equals the other argument.

The operator LCM computes the least common multiple of its two arguments. The result is always nonnegative. If either argument is 1, the result equals the other argument. If either argument is 0, the result is 0.

The type $\mathbb{Z}_{>}$ extends the trait PartialOrderAndMeetBoundedLattice[[$\mathbb{Z}_{>}$, |, GCD, LCM]], which is to say that the strictly positive integers form a partial order and lattice with the “evenly divides” operator | as the partial order operator and with GCD and LCM as the lattice operators.

property $\forall(m, n) ((m \text{ GCD } n) | m) \wedge ((m \text{ GCD } n) | n)$
property $\forall(m, n) (m | (m \text{ LCM } n)) \wedge (n | (m \text{ LCM } n))$
property $\forall(m, n) (m \text{ GCD } n) \cdot (m \text{ LCM } n) = m \cdot n$

25.2.44 opr (*self*)!: \mathbb{N}

The factorial operator is defined only for natural number types; it returns the product of all positive integers that are not less than the integer. If the argument is 0, the result is 1. For type \mathbb{N}^* , if the argument is $+\infty$, the result is $+\infty$.

property $\forall(m) m! = \prod_{k \leftarrow 1:m} k$

25.2.45 opr CHOOSE(*self*, *other*: \mathbb{Z}): \mathbb{Z}

The CHOOSE operator is defined only for natural number types; it computes the binomial coefficient $m \text{ CHOOSE } n = m! / (n!(m - n)!)$. If $n < 0$ or $n > m$, the result is 0.

property $\forall(m, n: \mathbb{Z}) (m \text{ CHOOSE } n) + (m \text{ CHOOSE } (n + 1)) = ((m + 1) \text{ CHOOSE } (n + 1))$

25.2.46 opr <(*self*, *other*: \mathbb{Z}): Boolean
25.2.47 opr ≤(*self*, *other*: \mathbb{Z}): Boolean
25.2.48 opr =(*self*, *other*: \mathbb{Z}): Boolean
25.2.49 opr ≥(*self*, *other*: \mathbb{Z}): Boolean
25.2.50 opr >(*self*, *other*: \mathbb{Z}): Boolean

The comparison operators <, ≤, =, ≥, and > allow any integer value to be compared numerically to any other integer value. Integer values, including $+\infty$ and $-\infty$, are totally ordered. The value $-\infty$ is less than any finite integer value, and $+\infty$ is greater than any finite integer value.

For \neq see Section 26.1.4.

25.2.51 opr `CMP`(`self`, `other`: \mathbb{Z}^*): TotalComparison

The `CMP` operator compares the arguments and returns one of the three values `LessThan`, `EqualTo`, and `GreaterThan`.

25.2.52 opr `MAX`(`self`, `other`: \mathbb{Z}): \mathbb{Z}

25.2.53 opr `MIN`(`self`, `other`: \mathbb{Z}): \mathbb{Z}

25.2.54 opr `MAXNUM`(`self`, `other`: \mathbb{Z}): \mathbb{Z}

25.2.55 opr `MINNUM`(`self`, `other`: \mathbb{Z}): \mathbb{Z}

The operators `MAX` and `MAXNUM` return whichever argument is larger in the total order defined by $<$, \leq , $=$, \geq , $>$, and `CMP`, and the operators `MIN` and `MINNUM` return whichever argument is smaller. (For all four, if the arguments are equal, then the result equals that same value.)

25.2.56 opr `|self|`: \mathbb{N}

The absolute value operator `|...|` returns the negative of this integer if the argument is less than zero, and otherwise returns the argument.

25.2.57 `signum`(`self`): \mathbb{Z}

The method `signum` returns -1 if this integer is less than zero, 0 if this integer is zero, and 1 if this integer is greater than zero.

25.2.58 `numerator`(`self`): \mathbb{Z}

25.2.59 `denominator`(`self`): \mathbb{Z}

For finite integers, the method `numerator` returns the argument and the method `denominator` returns 1 .

For type \mathbb{Z}^* , the numerator of $+\infty$ is 1 , and the numerator of $-\infty$ is -1 ; the denominator of either $+\infty$ or $-\infty$ is 0 .

25.2.60 `floor`(`self`): \mathbb{Z}

25.2.61 opr `[self]`: \mathbb{Z}

25.2.62 `ceiling`(`self`): \mathbb{Z}

25.2.63 opr `ceil`(`self`): \mathbb{Z}

25.2.64 `round`(`self`): \mathbb{Z}

25.2.65 `truncate`(`self`): \mathbb{Z}

These methods and operators simply return the argument when applied to an integer.

25.2.66 `opr` $\lfloor\!\lfloor\text{self}\!\!\rfloor\rfloor$: \mathbb{N}
25.2.67 `opr` $\lceil\!\lceil\text{self}\!\!\rceil\rceil$: \mathbb{N}
25.2.68 `opr` $\lfloor\!\!\lfloor\text{self}\!\!\rfloor\rfloor$: \mathbb{N}
25.2.69 `opr` $\lceil\!\!\lceil\text{self}\!\!\rceil\rceil$: \mathbb{N}

The hyperfloor operation $\lfloor\!\lfloor x\!\!\rfloor\rfloor$ computes $2^{\lfloor\log_2 x\rfloor}$ and returns the result as a natural number. If the argument is equal to 0, the result is 0. If the argument is negative, an `InvalidArgumentException` is thrown.

The hyperceiling operation $\lceil\!\lceil x\!\!\rceil\rceil$ computes $2^{\lceil\log_2 x\rceil}$ and returns the result as a natural number. If the argument is equal to 0, the result is 0. If the argument is negative, an `InvalidArgumentException` is thrown.

The hyperhyperfloor operation $\lfloor\!\!\lfloor x\!\!\rfloor\rfloor$ computes $2^{\lfloor\log_2 x\rfloor}$ and returns the result as a natural number. If the argument is equal to 0 or 1, the result is the same as the argument. If the argument is negative, an `InvalidArgumentException` is thrown.

The hyperhyperceiling operation $\lceil\!\!\lceil x\!\!\rceil\rceil$ computes $2^{\lceil\log_2 x\rceil}$ and returns the result as a natural number. If the argument is equal to 0 or 1, the result is the same as the argument. If the argument is negative, an `InvalidArgumentException` is thrown.

25.2.70 `shift(self, k: IndexInt): \mathbb{Z}`

The result of `shift(x, k)` is $\lfloor x \cdot 2^k \rfloor$. This corresponds to what is sometimes called an “arithmetic shift” on the two’s-complement representation of the integer; positive values of k shifts the bits to the left, and negative values of k shifts the bits to the right.

25.2.71 `bit(self, k: IndexInt): Bit`

The result of `bit(x, k)` is $\lfloor x \cdot 2^{-k} \rfloor \text{ MOD } 2$, as a Bit (0 or 1). If this integer is regarded as represented in binary two’s-complement form, with bits numbered in “little-endian order” (least significant bit is bit number 0, least significant bit but one is bit number 1, and so on), then this functional method returns bit k of this representation.

25.2.72 `opr` $\neg(\text{self})$: \mathbb{Z}

If this integer is regarded as represented in binary two’s-complement form, then this functional method returns an integer that is the bitwise complement of this integer—every bit is inverted.

`property` $\forall(x, k: \text{IndexInt}) \text{ bit}(\neg x, k) = \neg \text{bit}(x, k)$

- 25.2.73** `opr` $\&$ (`self`, `other`: \mathbb{Z}): \mathbb{Z}
25.2.74 `opr` $\&$ (`self`, `other`: \mathbb{Z}): \mathbb{Z}
25.2.75 `opr` $\underline{\&}$ (`self`, `other`: \mathbb{Z}): \mathbb{Z}

If the arguments are regarded as represented in binary two's-complement form, then these functional methods each return an integer that is the result of bitwise operations (*and*, *or*, or *exclusive or*, respectively) on the argument integers.

`property` $\forall(x, y, k: \text{IndexInt}) \text{bit}(x \& y, k) = \text{bit}(x, k) \wedge \text{bit}(y, k)$
`property` $\forall(x, y, k: \text{IndexInt}) \text{bit}(x \& y, k) = \text{bit}(x, k) \vee \text{bit}(y, k)$
`property` $\forall(x, y, k: \text{IndexInt}) \text{bit}(x \underline{\&} y, k) = \text{bit}(x, k) \underline{\vee} \text{bit}(y, k)$

- 25.2.76** `countBits`(`self`): `IndexInt`

If this integer is regarded as represented in binary two's-complement form, then this functional method returns the number of 1-bits in the representation if this integer is nonnegative, but returns the number of 0-bits in the representation if this integer is negative.

`property` $\forall(x) \text{countBits}(0) = 0$
`property` $\forall(x) \text{countBits}(x) = \text{countBits}(\neg x)$
`property` $\forall(x) \text{countBits}(\text{shift}(x, 1)) = \text{countBits}(x)$
`property` $\forall(x) \text{countBits}(\text{shift}(x, 1)) + 1 = \text{countBits}(x) + 1$

- 25.2.77** `countFactorsOfTwo`(`self`): `IndexInt`

This method returns the number of factors of 2 in this integer, that is, the logarithm of the largest power of 2 that even divides this integer. If this integer is regarded as represented in binary two's-complement form, then this method returns the number of trailing 0-bits. If this integer is zero, this method throws an `IntegerOverflowException`.

`property` $\forall(x) \text{countFactorsOfTwo}(1) = 0$
`property` $\forall(x) x \neq 0 \rightarrow \text{countFactorsOfTwo}(x) = \text{countFactorsOfTwo}(-x)$
`property` $\forall(x) x \neq 0 \rightarrow \text{countFactorsOfTwo}(\text{shift}(x, 1)) = \text{countFactorsOfTwo}(x) + 1$
`property` $\forall(x) \text{countFactorsOfTwo}(\text{shift}(x, 1)) + 1 = 0$

- 25.2.78** `integerLength`(`self`): `IndexInt`

This method returns $\lceil \log_2(-\text{self}) \rceil$ if `self` is negative, but returns $\lceil \log_2(\text{self} + 1) \rceil$ if `self` is nonnegative. Suppose that this integer is regarded as represented in binary two's-complement form, and this method returns the value k ; then k is the smallest integer such that $k + 1$ bits suffice to represent this integer in binary two's-complement form. Moreover, if this integer is nonnegative, then k is the smallest integer such that k bits suffice to represent this integer in unsigned binary form.

`property` $\forall(x) \text{integerLength}(0) = 0$
`property` $\forall(x) x \geq 0 \rightarrow \text{integerLength}(\text{shift}(x, 1)) = \text{integerLength}(\text{shift}(x, 1) + 1) = \text{integerLength}(x) + 1$
`property` $\forall(x) \text{integerLength}(x) = \text{integerLength}(\neg x)$

25.2.79 *lowBits*(self, k: IndexInt): \mathbb{N}

This method returns the value of this integer modulo 2^k . If this integer is regarded as represented in binary two's-complement form, then this method returns an integer whose k least significant bits are equal to the corresponding bits of this integer, and whose other bits are all 0.

property $\forall(x, k: \text{IndexInt}) \text{lowBits}(x, k) = x \text{ MOD } 2^k$

25.2.80 *even*(self): Boolean

25.2.81 *odd*(self): Boolean

The method *even* returns *true*, and the method *odd* returns *false*, if this integer is evenly divisible by two. The method *even* returns *false*, and the method *odd* returns *true*, if this integer is not evenly divisible by two. For type \mathbb{Z}^* , if this integer is $+\infty$ or $-\infty$, an *InvalidArgumentException* is thrown.

property $\forall(a) \text{even } a \leftrightarrow 2 \mid a$
property $\forall(a) \text{odd } a \leftrightarrow \neg \text{even } a$

25.2.82 *prime*(self): Boolean

The method *prime* is defined only for natural number types; it returns *true* if this integer is a prime number, and otherwise returns *false*. It returns *false* if this integer is 0 or 1.

$\forall(a, b) (a > 1) \wedge (a \mid b) \rightarrow: \neg \text{prime } b$

25.2.83 *realpart*(self): \mathbb{Z}

The method *realpart* for an integer simply returns its argument.

25.2.84 *imagpart*(self): \mathbb{Z}

The method *imagpart* for an integer simply returns zero.

- 25.2.85** `check(self): \mathbb{Z}` throws `CastException`
- 25.2.86** `check*(self): \mathbb{Z}^*` throws `CastException`
- 25.2.87** `check<(self): $\mathbb{Z}_{<}$` throws `CastException`
- 25.2.88** `check \leq (self): \mathbb{Z}_{\leq}` throws `CastException`
- 25.2.89** `check \geq (self): \mathbb{Z}_{\geq}` throws `CastException`
- 25.2.90** `check>(self): $\mathbb{Z}_{>}$` throws `CastException`
- 25.2.91** `check \neq (self): \mathbb{Z}_{\neq}` throws `CastException`
- 25.2.92** `check* \leq (self): \mathbb{Z}_{\leq}^*` throws `CastException`
- 25.2.93** `check* \geq (self): \mathbb{Z}_{\geq}^*` throws `CastException`
- 25.2.94** `check* $>$ (self): $\mathbb{Z}_{>}^*$` throws `CastException`
- 25.2.95** `check* \neq (self): \mathbb{Z}_{\neq}^*` throws `CastException`
- 25.2.96** `check* \neq (self): \mathbb{Z}_{\neq}^*` throws `CastException`

Each of these methods checks this integer to see whether it belongs to the result type of the method. If, the number is returned; if not, a `CastException` is thrown.

Future versions of this specification will include trait declarations definitions for the other numeric quantities.

Chapter 26

Negated Relational Operators

26.1 Negated Equivalence Operators

- 26.1.1 $\text{opr } \not\equiv(x: \text{Any}, y: \text{Any}): \text{Boolean}$
- 26.1.2 $\text{opr } \not\equiv[[T \text{ extends BinaryPredicate}[[T, \equiv]]](x: T, y: T): \text{Boolean}$
- 26.1.3 $\text{opr } \not\equiv[[T \text{ extends BinaryIntervalPredicate}[[T, \equiv]]](x: T, y: T): \text{BooleanInterval}$
- 26.1.4 $\text{opr } \neq[[T \text{ extends BinaryPredicate}[[T, =]]](x: T, y: T): \text{Boolean}$
- 26.1.5 $\text{opr } \neq[[T \text{ extends BinaryIntervalPredicate}[[T, =]]](x: T, y: T): \text{BooleanInterval}$
- 26.1.6 $\text{opr } \not\approx[[T \text{ extends BinaryPredicate}[[T, \simeq]]](x: T, y: T): \text{Boolean}$
- 26.1.7 $\text{opr } \not\approx[[T \text{ extends BinaryIntervalPredicate}[[T, \simeq]]](x: T, y: T): \text{BooleanInterval}$
- 26.1.8 $\text{opr } \not\approx[[T \text{ extends BinaryPredicate}[[T, \approx]]](x: T, y: T): \text{Boolean}$
- 26.1.9 $\text{opr } \not\approx[[T \text{ extends BinaryIntervalPredicate}[[T, \approx]]](x: T, y: T): \text{BooleanInterval}$

The infix operator $\not\equiv$ applies \neg to the result of \equiv on the same operands.

The infix operator \neq applies \neg to the result of $=$ on the same operands.

The infix operator $\not\approx$ applies \neg to the result of \simeq on the same operands.

The infix operator $\not\approx$ applies \neg to the result of \approx on the same operands.

26.2 Negated Comparison Operators

- 26.2.1** $\text{opr } \not< \llbracket T \text{ extends BinaryPredicate} \llbracket T, < \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.2** $\text{opr } \not< \llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, < \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$
- 26.2.3** $\text{opr } \not\leq \llbracket T \text{ extends BinaryPredicate} \llbracket T, \leq \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.4** $\text{opr } \not\leq \llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, \leq \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$
- 26.2.5** $\text{opr } \not\geq \llbracket T \text{ extends BinaryPredicate} \llbracket T, \geq \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.6** $\text{opr } \not\geq \llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, \geq \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$
- 26.2.7** $\text{opr } \not> \llbracket T \text{ extends BinaryPredicate} \llbracket T, > \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.8** $\text{opr } \not> \llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, > \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$

The infix operator $\not<$ applies \neg to the result of $<$ on the same operands.

The infix operator $\not\leq$ applies \neg to the result of \leq on the same operands.

The infix operator $\not\geq$ applies \neg to the result of \geq on the same operands.

The infix operator $\not>$ applies \neg to the result of $>$ on the same operands.

- 26.2.9** $\text{opr } \not\subset \llbracket T \text{ extends BinaryPredicate} \llbracket T, \subset \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.10** $\text{opr } \not\subset \llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, \subset \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$
- 26.2.11** $\text{opr } \not\subseteq \llbracket T \text{ extends BinaryPredicate} \llbracket T, \subseteq \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.12** $\text{opr } \not\subseteq \llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, \subseteq \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$
- 26.2.13** $\text{opr } \not\supseteq \llbracket T \text{ extends BinaryPredicate} \llbracket T, \supseteq \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.14** $\text{opr } \not\supseteq \llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, \supseteq \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$
- 26.2.15** $\text{opr } \not\supset \llbracket T \text{ extends BinaryPredicate} \llbracket T, \supset \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.16** $\text{opr } \not\supset \llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, \supset \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$

The infix operator $\not\subset$ applies \neg to the result of \subset on the same operands.

The infix operator $\not\subseteq$ applies \neg to the result of \subseteq on the same operands.

The infix operator $\not\supseteq$ applies \neg to the result of \supseteq on the same operands.

The infix operator $\not\supset$ applies \neg to the result of \supset on the same operands.

- 26.2.17** opr $\not\prec$ $\llbracket T \text{ extends BinaryPredicate} \llbracket T, \prec \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.18** opr $\not\prec$ $\llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, \prec \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$
- 26.2.19** opr \preceq $\llbracket T \text{ extends BinaryPredicate} \llbracket T, \preceq \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.20** opr \preceq $\llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, \preceq \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$
- 26.2.21** opr \succcurlyeq $\llbracket T \text{ extends BinaryPredicate} \llbracket T, \succcurlyeq \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.22** opr \succcurlyeq $\llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, \succcurlyeq \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$
- 26.2.23** opr \succ $\llbracket T \text{ extends BinaryPredicate} \llbracket T, \succ \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.24** opr \succ $\llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, \succ \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$

The infix operator $\not\prec$ applies \neg to the result of \prec on the same operands.

The infix operator \preceq applies \neg to the result of \preceq on the same operands.

The infix operator \succcurlyeq applies \neg to the result of \succcurlyeq on the same operands.

The infix operator \succ applies \neg to the result of \succ on the same operands.

- 26.2.25** opr \sqsubset $\llbracket T \text{ extends BinaryPredicate} \llbracket T, \sqsubset \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.26** opr \sqsubset $\llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, \sqsubset \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$
- 26.2.27** opr \sqsubseteq $\llbracket T \text{ extends BinaryPredicate} \llbracket T, \sqsubseteq \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.28** opr \sqsubseteq $\llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, \sqsubseteq \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$
- 26.2.29** opr \supseteq $\llbracket T \text{ extends BinaryPredicate} \llbracket T, \supseteq \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.30** opr \supseteq $\llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, \supseteq \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$
- 26.2.31** opr \supset $\llbracket T \text{ extends BinaryPredicate} \llbracket T, \supset \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.32** opr \supset $\llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, \supset \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$

The infix operator \sqsubset applies \neg to the result of \sqsubset on the same operands.

The infix operator \sqsubseteq applies \neg to the result of \sqsubseteq on the same operands.

The infix operator \supseteq applies \neg to the result of \supseteq on the same operands.

The infix operator \supset applies \neg to the result of \supset on the same operands.

- 26.2.33** opr \notin $\llbracket T \text{ extends BinaryPredicate} \llbracket T, \in \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.34** opr \notin $\llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, \in \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$
- 26.2.35** opr \ni $\llbracket T \text{ extends BinaryPredicate} \llbracket T, \ni \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.36** opr \ni $\llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, \ni \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$

The infix operator \notin applies \neg to the result of \in on the same operands.

The infix operator \ni applies \neg to the result of \ni on the same operands.

- 26.2.37** opr \parallel $\llbracket T \text{ extends BinaryPredicate} \llbracket T, \parallel \rrbracket \rrbracket (x: T, y: T): \text{Boolean}$
- 26.2.38** opr \parallel $\llbracket T \text{ extends BinaryIntervalPredicate} \llbracket T, \parallel \rrbracket \rrbracket (x: T, y: T): \text{BooleanInterval}$

The infix operator \parallel applies \neg to the result of \parallel on the same operands.

Chapter 27

Exceptions

27.1 The Trait `Fortress.Standard.Exception`

The trait `Exception` is a single root of the exception hierarchy; every exception in Fortress has trait `Exception`. An exception is either a `CheckedException` or an `UncheckedException`. Every exception has optional fields: a `message` and a chained exception. These fields are default to `Nothing` where an optional value v is either `Nothing` or `Just(v)` as declared in Section 31.2.

```
trait Exception comprises { CheckedException, UncheckedException }
  settable message: Maybe[String]
  settable chain: Maybe[Exception]
end
```

27.1.1 `settable message: Maybe[String]`

When an exception is thrown, its `message` may be set.

27.1.2 `settable chain: Maybe[Exception]`

When an exception is thrown, its `chain` may be set to the exception thrown immediately before this exception.

27.2 The Trait `Fortress.Standard.CheckedException`

```
trait CheckedException
  extends { Exception }
  excludes { UncheckedException }
end
```

27.3 The Trait `Fortress.Standard.UncheckedException`

```
trait UncheckedException
  extends { Exception }
  excludes { CheckedException }
end
```

Chapter 28

Threads

28.1 The Trait `Fortress.Standard.Thread`

Every thread in Fortress has trait `Thread`.

```
trait Thread[[T extends Any]]
  val(): T
  wait(): ()
  ready(): Boolean
  stop(): () throws Stopped
end
```

28.1.1 `val(): T`

The `val` method returns the value computed by the expression of the thread. If the thread has not yet completed execution, the invocation of `val` blocks until it has done so.

28.1.2 `wait(): ()`

The `wait` method waits for a thread to complete, but does not return a value.

28.1.3 `ready(): Boolean`

The `ready` method returns `true` if a thread has completed, and returns `false` otherwise.

28.1.4 `stop(): () throws Stopped`

The `stop` method attempts to terminate a thread.

Chapter 29

Dimensions and Units

29.1 Fortress.SIUnits

(* Reference: <http://physics.nist.gov/cuu/Units/index.html> *)

(* SI base units *)

dim Length SI_unit meter meters m
dim Mass default kilogram; SI_unit gram grams g; Mass
dim Time SI_unit second seconds s
dim ElectricCurrent SI_unit ampere amperes A
dim Temperature SI_unit kelvin kelvins K
dim AmountOfSubstance SI_unit mole moles mol
dim LuminousIntensity SI_unit candela candelas cd

(* SI derived units with special names and symbols *)

dim Angle = Unity SI_unit radian radians rad
dim SolidAngle = Unity SI_unit steradian steradians sr
dim Frequency = 1/Time SI_unit hertz Hz
dim Force = Mass Acceleration SI_unit newton newtons N
dim Pressure = Force/Area SI_unit pascal pascals Pa
dim Energy = Length Force SI_unit joule joules J
dim Power = Energy/Time SI_unit watt watts W
dim ElectricCharge = ElectricCurrent Time SI_unit coulomb coulombs C
dim ElectricPotential = Power/Current SI_unit volt volts V
dim Capacitance = ElectricCharge/Voltage SI_unit farad farads F
dim Resistance = ElectricPotential/Current SI_unit ohm ohms Ω
dim Conductance = 1/Resistance SI_unit siemens S
dim MagneticFlux = Voltage Time SI_unit weber webers Wb
dim MagneticFluxDensity = MagneticFlux/Area SI_unit tesla teslas T
dim Inductance = MagneticFlux/Current SI_unit henry henries H
dim LuminousFlux = LuminousIntensity SolidAngle SI_unit lumen lumens lm
dim Illuminance = LuminousFlux/Area SI_unit lux lx
dim RadionuclideActivity = 1/Time SI_unit becquerel becquerels Bq
dim AbsorbedDose = Energy/Mass SI_unit gray grays Gy
dim CatalyticActivity = AmountOfSubstance/Time SI_unit katal katal kat

(* Other derived dimensions *)

```

dim Area = Length2
dim Volume = Length3
dim Velocity = Length/Time
dim Speed = Velocity
dim Acceleration = Velocity/Time
dim Momentum = Mass Velocity
dim AngularVelocity = Angle/Second
dim AngularAcceleration = Angle/Second2
dim WaveNumber = 1/Length
dim MassDensity = Mass/Volume
dim CurrentDensity = Current/Area
dim MagneticFieldStrength = Current/Length
dim Luminance = LuminousIntensity/Area
dim Work = Energy
dim Action = Energy Time
dim MomentOfForce = Force Length
dim Torque = MomentOfForce
dim MomentOfInertia = Mass Length2
dim Voltage = ElectricPotential
dim Conductivity = Conductance/Length
dim Resistivity = 1/Conductivity
dim Impedance = Resistance
dim Permittivity = Capacitance/Length
dim Permeability = Inductance/Length
dim Irradiance = Power/Area
dim RadiantIntensity = Power/SolidAngle
dim Radiance = Power/Area SolidAngle
dim AbsorbedDoseRate = AbsorbedDose/Time
dim CatalyticConcentration = CatalyticActivity/Volume
dim HeatCapacity = Energy/Temperature
dim Entropy = Energy/Temperature
dim DynamicViscosity = Pressure Time
dim SpecificHeatCapacity = Energy/Mass Temperature
dim SpecificEntropy = Energy/Mass Temperature
dim SpecificEnergy = Energy/Mass
dim ThermalConductivity = Energy/Length Temperature
dim EnergyDensity = Energy/Volume
dim ElectricFieldStrength = ElectricPotential/Length
dim ElectricChargeDensity = ElectricCharge/Volume
dim ElectricFlux = ElectricCharge
dim ElectricFluxDensity = ElectricCharge/Area
dim MolarEnergy = Energy/AmountOfSubstance
dim MolarHeatCapacity = Energy/AmountOfSubstance Temperature
dim MolarEntropy = Energy/AmountOfSubstance Temperature
dim RadiationExposure = ElectricCharge/Mass

```

(* Units outside the SI that are accepted for use with the SI *)

```

unit minute minutes min: Time
unit hour hours h: Time
unit day days d: Time
unit degreeOfAngle degrees: Angle
unit minuteOfAngle minutesOfAngle: Angle

```



```
unit secondOfAngle secondsOfAngle: Angle
SI_unit metricTon metricTons tonne tonnes t: Mass
SI_unit liter liters L: Volume
```

29.2 Fortress.EnglishUnits

```
import { Length, Area, Volume, Time, Mass, millimeters, liters, grams }
    from Fortress.SIUnits

unit inch inches: Length
unit foot feet: Length
unit yard yards: Length
unit mile miles: Length
unit rod rods: Length
unit furlong furlongs: Length

unit surveyFoot surveyFeet: Length
unit surveyMile surveyMiles: Length

unit nauticalMile nauticalMiles: Length
unit knot knots: Speed

unit week weeks: Time
unit fortnight fortnights: Time
unit microfortnight microfortnights: Time

unit gallon gallons: Volume
unit fluidQuart fluidQuarts: Volume
unit fluidPint fluidPints: Volume
unit fluidCup fluidCups: Volume
unit fluidOunce fluidOunces: Volume
unit fluidDram fluidDrams: Volume
unit minim minims: Volume

unit traditionalTablespoon traditionalTablespoons: Volume
unit traditionalTeaspoon traditionalTeaspoons: Volume
unit federalTablespoon federalTablespoons: Volume
unit federalTeaspoon federalTeaspoons: Volume

unit dryPint dryPints: Volume
unit dryQuart dryQuarts: Volume
unit peck pecks: Volume
unit bushel bushels: Volume

unit acre: Area

unit imperialGallon: Volume
unit imperialQuart: Volume
unit imperialPint: Volume
unit imperialGill: Volume
unit imperialFluidOunce: Volume
unit imperialFluidDrachm: Volume
unit imperialFluidDRam : Volume
unit imperialFluidScruple: Volume
unit imperialMinim: Volume

unit pound pounds lb lbs: Mass
```

`unit` ounce ounces oz: Mass
`unit` grain grains: Mass
`unit` troyPound troyPounds: Mass
`unit` troyOunce troyOunces: Mass

29.3 Fortress.InformationUnits

`dim` Information `unit` bit bits
`unit` byte bytes: Information

Chapter 30

Tests

30.1 The Object `Fortress.Standard.TestSuite`

An instance of the object `TestSuite` contains a set of test functions that can all be called by invoking the method `run`:

```
test object TestSuite(testFunctions = {})
  add(f: () → ()): ()
  run(): ()
end
```

30.1.1 `add(f: () → ()): ()`

The `add` method adds a given test function to the `testFunctions` field of this object.

30.1.2 `run(): ()`

The `run` method calls each test function in the `testFunctions` field of this object. Note that all tests in a `TestSuite` are run in parallel.

30.2 Test Functions

30.2.1 `test fail(message: String): ()`

The helper function `fail` displays the error message provided and terminates execution of the enclosing test.

Chapter 31

Convenience Functions and Types

31.1 Convenience Functions

31.1.1 `cast[[T extends Any]](x: Any): T`

The function `cast` converts the type of its argument to a given type. If the static type of the argument is not a subtype of the given type, a `CastException` is thrown.

31.1.2 `instanceOf[[T extends Any]](x: Any): Boolean`

The function `instanceOf` tests whether its argument has a given type and returns a boolean value.

31.1.3 `ignore(x: Any): ()`

The function `ignore` discards the value of its argument and returns `()`.

31.1.4 `tuple[[T extends Tuple]](x: T): T`

The function `tuple` returns its argument as a tuple expression.

31.1.5 `identity[[T extends Any]](x: T): T`

The function `identity` returns its argument.

31.1.6 `coerce[[T]](x: T): T`

The function `coerce` returns its argument as the given type.

31.2 Convenience Types

An optional value v is either `Nothing` or `Just(v)` declared as follows:

```
(* Optional Values *)
trait Maybe[[T]] comprises { Nothing, Just[[T]] }
  isNothing: Boolean
end
object Nothing extends Maybe[[T]] excludes Just[[T]] where {T extends Object}
end
object Just[[T]](just: T) extends Maybe[[T]]
end
```

Part IV

Fortress for Library Writers

Chapter 32

Parallelism and Locality

Fortress is designed to make parallel programming as simple and as painless as possible. This chapter describes the internals of Fortress parallelism designed for use by authors of library code (such as *distributions*, generators, and arrays). We adopt a multi-tiered approach to parallelism:

- At the highest level, we provide libraries which allocate locality-aware distributed arrays (Section 32.2) and implicitly parallel constructs such as tuples and loops. Synchronization is accomplished through the use of atomic sections (Section 13.22). More complex synchronization makes use of abortable atomicity, described in Section 32.3.
- There is an extensive library of distributions, which permits the programmer to specify locality and data distribution explicitly (Section 32.5).
- Immediately below that, the `at` expression requests that a computation take place in a particular region of the machine (Section 32.7). We also provide a mechanism to terminate a spawned thread early (Section 32.6).
- Finally, there are mechanisms for constructing new generators via recursive subdivision into tree structures with individual elements at the leaves. Section 32.8 explains how iterative constructs such as `for` loops and comprehensions are desugared into calls to methods of trait `Generator`, and how new instances of this trait may be defined.

We begin by describing the abstraction of *regions*, which Fortress uses to describe the machine on which a program is run.

32.1 Regions

Every thread (either explicit or implicit) and every object in Fortress, and every element of a Fortress array (the physical storage for that array element), has an associated *region*. The region in which an object *o* resides can be obtained by calling `o.region`. Regions abstractly describe the structure of the machine on which a Fortress program is running. They are organized hierarchically to form a tree, the *region hierarchy*, reflecting in an abstract way the degree of locality which those regions share. The different levels of this tree reflect underlying machine structure, such as execution engines within a CPU, memory shared by a group of processors, or resources distributed across the entire machine. Objects which reside in regions near the leaves of the tree are local entities; those which reside at higher levels of the region tree are logically spread out. The method call `r.isLocalTo(s)` returns `true` if *r* is contained within the region tree rooted at *s*.

It is important to understand that regions and the structures (such as distributions, Section 32.5) built on top of them exist purely for performance purposes. The placement of a thread or an object does not have any semantic effect on

the meaning of a program; it is simply an aid to enable the implementation to make informed decisions about data placement.

It may not be possible for an object or a thread to reside in a given region. Threads of execution reside at the *execution level* of the region hierarchy, generally the bottommost level in the region tree. Each thread is generally associated with some region at the execution level, indicating where it will preferentially be run. The programmer can affect the choice of region by using an `at` expression (Section 32.7) when the thread is created. A thread may have an associated region which is not at the execution level of the region hierarchy, either because a higher region was requested with an `at` expression or because scheduling decisions permit the thread to run in several possible execution regions. The region to which a thread is assigned may also change over time due to scheduling decisions. The *region* method for the object associated with a spawned thread returns the region of the associated thread.

The *memory level* of the region hierarchy is where individual reference objects reside; on a machine with nodes composed of multiple processor cores sharing a single memory, this generally will not be a leaf of the region hierarchy. Imagine a constructor for a reference object is called by a thread residing in region r , yielding an object o . Except in very rare circumstances (for example when a local node is out of memory) either $r.isLocalTo(o.region)$ or $(o.region).isLocalTo(r)$ ought to hold: data is allocated locally to the thread which runs the constructor. For a value object v being manipulated by a thread residing in region r either $(v.region).isLocalTo(r)$ or $r.isLocalTo(v.region)$ (value objects always appear to be local).

Note that *region* is a getter method and can be overridden like any other method. The chief example of this is arrays, which are generally composed from many reference objects; the *region* method is overridden to return the location of the array as a whole—the region which contains all of its constituent reference objects.

32.2 Distributed Arrays

Arrays, vectors, and matrices in Fortress are assumed to be spread out across the machine. As in Fortran, Fortress arrays are complex data structures; simple linear storage is encapsulated by the `HeapSequence` type, which is used in the implementation of arrays (see Section 32.7). The default distribution of an array is determined by the Fortress libraries; in general it depends on the size of the array, and on the size and locality characteristics of the machine running the program. For advanced users, the distribution library (introduced in Section 32.5) provides a way of combining and pivoting distributions, or of redistributing two arrays so that their distributions match. Programmers should create arrays by using an array comprehension (Section 13.28) or an aggregate expression (Section 13.27). The operational internals of array comprehensions are described in Section 32.8.

Because the elements of a fortress array may reside in multiple regions of the machine, there is an additional method $a.region(i)$ which returns the region in which the array element a_i resides. An element of an array is always local to the region in which the array as a whole is contained, so $(a.region(i)).isLocalTo(a.region)$ must always return *true*. When an array contains reference objects, the programmer must be careful to distinguish the region in which the array element a_i resides, $a.region(i)$, from the region in which the object referred to by the array element resides, $a_i.region$. The former describes the region of the array itself; the latter describes the region of the data referred to by the array. These may differ.

32.3 Abortable Atomicity

Fortress provides a user-level `abort()` function which abandons execution of an `atomic` expression and rolls back its changes, requiring the `atomic` expression to execute again from the beginning. This permits an atomic section to perform consistency checks as it runs. However, the functionality provided by `abort()` can be abused; it is possible to induce deadlock or livelock by creating an atomic section which always fails. Here is a simple example of a program using `abort()` which is incorrect because Fortress does not guarantee that the two implicit threads (created

by evaluating the two elements of the tuple) will always run in parallel; it is possible for the first element of the tuple to continually abort without ever running the second element of the tuple:

```
r : ℤ64 := 0
(a, b) = (atomic if r = 1 then 17 else abort() end,
         do r := 1; r end)(* INCORRECT! *)
```

Fortress also includes a `tryatomic` expression, which attempts to run its body expression atomically. If it succeeds, the result is returned; if it aborts due to a call to `abort`, the `AtomicAborted` exception is thrown; if it aborts due to conflict (as described in Section 13.22), the `AtomicConflict` exception is thrown. These exceptions both implement the `AtomicFailed` trait, which is an instance of `CheckedException`. Conceptually `atomic` can be defined in terms of `tryatomic` as follows:

```
label AtomicBlock
  while true do
    try
      result = tryatomic body
      exit AtomicBlock with result
    catch e
      AtomicFailed ⇒ ()(* continue execution *)
    end
  end
end
throw UnreachableCode(* inserted for type correctness *)
end AtomicBlock
```

Unlike the above definition, an implementation may choose to suspend a thread running an `atomic` expression which invokes `abort`, re-starting it at a later time when it may be possible to make further progress. The above definition restarts the body of the `atomic` expression immediately without suspending.

32.4 Shared and Local Data

Every object in a Fortress program is considered to be either *shared* or *local* (collectively referred to as the *sharedness* of the object). A local object must be transitively reachable (through zero or more object references) from the variables of at most one running thread. A local object may be accessed more cheaply than a shared object, particularly in the case of atomic reads and writes. Sharedness is ordinarily managed implicitly by the Fortress implementation. Control over sharedness is intended to be a performance optimization; however, methods such as `isShared` and `localize` can affect program semantics, and must be used with care.

The sharedness of an object should be contrasted with its region. The region of an object describes where that object is located on the machine. The sharedness of an object describes whether the object is visible to one thread or to many. A local object need not actually reside in a region near the thread to which it is visible (though ordinarily it will).

The following rules govern sharedness:

- Reference objects are initially local when they are constructed.
- The sharedness of an object may change as the program executes.
- If an object is currently transitively reachable from more than one running thread, it must be shared.
- When a reference to a local object is stored into a field of a shared object, the local object must be *published*: Its sharedness is changed to shared, and all of the data to which it refers is also published.
- The value of a local variable referenced by a thread must be published before that thread may be run in parallel with the thread which created it. Values assigned to the variable while the threads run in parallel must also be published.

- A field with value type is assigned by copying, and thus has the sharedness of the containing object or closure.

Publishing can be expensive, particularly if the structure being broadcast is large and heavily nested; this can cause an apparently short `atomic` expression (a single write, say) to run arbitrarily long. To avoid this, the library programmer can request that an object be published by calling the semantically transparent function `shared`:

```
x := shared Cons(x, xs)
shared(y)
```

A local copy of an object can be obtained by calling `copy`, a method on trait `Any`:

```
localVar := sharedVar.copy()
```

Two additional methods are provided which permit different choices of program behavior based on the sharedness of objects:

- The getter `o.isShared` returns `true` when `o` is shared, and `false` when it is local. This permits the program to take different actions based on sharedness.
- Method `o.localize()` is equivalent to the following expression:

```
if o.isShared then o.copy() else o end
```

These methods must be used with extreme caution. For example, `localize` should be used only when there is a unique reference to the object being localized. The `localize` method can have unexpected behavior if there is a reference to `o` from another local object `p`. Updates to `o` will be visible through `p`; subsequent publication of `p` will publish `o`. By contrast, if `o` was already shared, and referred to by another shared object, the newly-localized copy will be entirely distinct; changes to the copy will not be visible through `p`, and publishing `p` will not affect the locality of the copy.

32.5 Distributions

Most of the heavy lifting in mapping threads and arrays to regions is performed by *distributions*. An instance of the trait `Distribution` describes the parallel structure of ranges and other numeric generators (such as the generators for the index space of an array), and provides for the allocation and distribution of arrays on the machine:

```
trait Distribution
  distribute[[E, B extends ArrayIndex]](Range[[B]]): Range[[B]]
  distribute[[E, B extends ArrayIndex]](a: Array[[E, B]]): Array[[E, B]] =
    distributeFromTo[[E, B]](a, a.distribution, self)
end
```

Abstractly, a `Distribution` acts as a transducer for generators and arrays. The `distribute` method applied to a multidimensional `Range` organizes its indices into the leaves of a tree whose inner nodes correspond to potential levels of parallelism and locality in the underlying computation, producing a fresh `Range` whose behavior as a `Generator` may differ from that of the passed-in `Range`. The `distribute` method applied to an array creates a copy of that array distributed according to the given distribution. This is specified in terms of a call to the overloaded function `distributeFromTo`. This permits the definition of specialized versions of this function for particular pairs of distributions.

The intention of distributions is to separate the task of data distribution and program correctness. That is, it should be possible to write and debug a perfectly acceptable parallel program using only the default data distribution provided by the system. Imposing a distribution on particular computations, or designing and implementing distributions from scratch, is a task best left for performance tuning, and one which should not affect the correctness of a working program.

There is a `DefaultDistribution` which is defined by the underlying system. This distribution is designed to be reasonably adaptable to different system scales and architectures, at the cost of some runtime efficiency. Arrays and generators that are not explicitly allocated through a distribution are given the `DefaultDistribution`.

We said in Section 13.15 that there is a generator, *indices*, associated with every array. This generator is distributed in the same way as the array itself. When we re-distribute an array, we also re-distribute the generator; thus *d.distribute(a.indices)* is equivalent to *(d.distribute(a)).indices*.

There are a number of built-in distributions:

<code>DefaultDistribution</code>	Name for distribution chosen by system.
<code>Sequential</code>	Sequential distribution. Arrays are allocated in one contiguous piece of memory.
<code>Local</code>	Equivalent to <code>Sequential</code> .
<code>Par</code>	Blocked into chunks of size 1.
<code>Blocked</code>	Blocked into roughly equal chunks.
<code>Blocked(n)</code>	Blocked into <i>n</i> roughly equal chunks.
<code>Subdivided</code>	Chopped into 2^k -sized chunks, recursively.
<code>Interleaved(d_1, d_2, \dots, d_n)</code>	The first <i>n</i> dimensions are distributed according to $d_1 \dots d_n$, with subdivision alternating among dimensions.
<code>Joined(d_1, d_2, \dots, d_n)</code>	The first <i>n</i> dimensions are distributed according to $d_1 \dots d_n$, subdividing completely in each dimension before proceeding to the next.

From these, a number of composed distributions are provided:

<code>Morton</code>	Bit-interleaved Morton order [22], recursive subdivision in all dimensions.
<code>Blocked(x_1, x_2, \dots, x_n)</code>	Blocked in <i>n</i> dimensions into chunks of size x_i in dimension <i>i</i> ; remaining dimensions (if any) are local.

To allocate an array which is local to a single thread (and most likely allocated in contiguous storage), the `Local` distribution can be used:

```
a = Local.distribute[1 0 0; 0 1 0; 0 0 1]
```

Other distributions can be requested in a similar way.

Distributions can be constructed and given names:

```
spatialDist = Blocked(n, n, 1)(* Pencils along the z axis *)
```

The system will lay out arrays with the same distribution in the same way in memory (as much as this is feasible), and will run loops with the same distribution in the same way (as much as this is feasible). By contrast, if we replace every occurrence of *spatialDist* by `Blocked(n, n, 1)`, this code will likely divide up arrays and ranges into the same-sized pieces as above, but these pieces need not be collocated.

32.6 Early Termination of Threads

As noted in Section 4.4, an implicit thread can be terminated if its group is going to throw an exception. Similarly, a spawned thread *t* may be terminated by calling *t.stop()*. A successful attempt to terminate a thread causes the thread to complete asynchronously. There is no guarantee that termination attempts will be prompt, or that they will occur at all; the implementation will make its best effort. If a thread completes normally or exceptionally before an attempt to terminate it succeeds, the result is retained and the termination attempt is simply dropped.

A termination attempt acts as if a special hidden *stop exception* is thrown in that thread. This exception cannot be thrown by `throw` or caught by `catch`; however, `finally` clauses are run as with any other exception. If the stopped thread was in the middle of an `atomic` expression, the effects of that expression are rolled back just as with an ordinary `throw`. A special wrapper around every spawned thread is provided by the Fortress implementation; it catches the

stop exception and transforms it into a deferred `Stopped` exception. This is visible to the programmer and should be caught by invoking the `val` method on the thread object. Implicit threads are terminated only if another thread in the group completes abruptly, and the threads that are terminated are ignored for the purposes of the completion of the group.

Typical code for stopping a thread looks something like the following example:

```
x : ℤ64 := 0
t = spawn do
  try
    atomic if x = 0 then abort() else () end
  finally
    x := 1
  end
end
t.stop()
try
  t.val()
catch s
  Stopped ⇒ x += 2; x
end
```

Here the spawned thread `t` blocks until it is killed by the call to `t.stop()`; it sets `x` to 1 in the `finally` clause before exiting. In this case, the call to `t.val()` will throw `Stopped`, which is caught, causing 2 to be added to `x` and returning 3.

Note that there is a race in the above code, so the `try` block in `t` may not have been entered when `t.stop()` is called, causing `x` to be 2 at the end of execution. Note also that the call to `t.stop()` occurs asynchronously; in the absence of the call to `t.val()`, the spawning thread would not have waited for `t` to complete.

32.7 Placing Threads

A thread can be placed in a particular region by using an `at` expression:

```
(v, w) = (ai,
  at a.region(j) do
    aj
  end)
```

In this example, two implicit threads are created; the first computes a_i locally, the second computes a_j in the region where the j^{th} element of a resides, specified by `a.region(j)`. The expression after `at` must return a value of type `Region`, and the block immediately following `do` is run in that region; the result of the block is the result of the `at` expression as a whole. Often it is more graceful to use the `also do` construct (described in Section 13.11.3) in these cases:

```
do
  v := ai
also at a.region(j) do
  w := aj
end
```

We can also use `at` with a `spawn` expression:

```

v = spawn at a.region(i) do
    a_i
end
w = spawn at v.region() do
    v.val() · 17
end

```

Finally, note that it is possible to use an `at` expression within a block:

```

do
    v := a_i
    at a.region(j) do
        w := a_j
    end
    x = v + w
end

```

We can think of this as the degenerate case of `also do`: a thread group is created with a single implicit thread running the contents of the `at` expression in the given region; when this thread completes control returns to the original location.

Note that the regions given in an `at` expression are non-binding: the Fortress implementation may choose to run the computations elsewhere—for example, thread migration might not be possible within an `atomic` expression, or load balancing might cause code to be executed in a different region. In general, however, implementations should attempt to respect thread placement annotations when they are given.

32.8 Use and Definition of Generators

Several expressions in Fortress make use of *generator lists* (given by the nonterminal *GeneratorList* in the Fortress grammar defined in Appendix G) to express parallel iteration (see Section 13.14). A generator list binds a series of variables to the values produced by a series of objects with the *Generator* trait. A generator list is simply syntactic sugar for a nested series of invocations of methods on these objects. *All the parallelism provided by a particular generator is specified by its definitions for the methods of the Generator trait.* In general, the library code for a generator dictates the parallel structure of computations involving that generator.

The definition of trait *Generator* has very simple functionality at its core:

```

trait Generator[E extends Any]
    size : Z64
    generate[R extends Monoid[R, ⊕], opr ⊕](body : E → R) : R
    join[N extends Any](other : Generator[N]) : Generator[(E, N)] =
        SimplePairGenerator[E, N](self, other)
end

```

The mechanics of object generation are embedded entirely in the *generate* method. This method takes one argument, the *body* function. The *generate* method invokes *body* once for each object which is to be generated, passing the generated object as an argument. Note that *body* returns a value in some Monoid *R*; the results of the calls to *body* are combined using the monoid operator \oplus . This *reduction* may include any number of occurrences of the identity of the monoid—in particular, a generator may generate no elements, in which case it will never invoke *body* and will simply the identity.

Note that every generator has a *natural order*: an order in which the terms generated by different executions of the *body* expression are combined by the reduction. This is important when the operator in question is associative but not commutative.

```

 $\mathcal{T}[[R, \boxplus][ ]]$       body = body
 $\mathcal{T}[[R, \boxplus][ x \leftarrow g, gs ]]$  body = g.generate[[R,  $\boxplus$ ]](fn ()  $\Rightarrow$   $\mathcal{T}[[R, \boxplus][gs]$ body)
 $\mathcal{T}[[R, \boxplus][ p, gs ]]$       body = if p then ( $\mathcal{T}[[R, \boxplus][gs]$ body) else Identity[[ $\boxplus$ ]] end

```

Figure 32.1: Naive and simple desugaring of generator lists using only the *generate* method.

A simple definition of a Generator need only define the *size* field and the *generate* method:

```

value object BlockedRange(lo:  $\mathbb{Z}64$ , hi:  $\mathbb{Z}64$ , b:  $\mathbb{Z}64$ ) extends Generator[[ $\mathbb{Z}64$ ]]
  size :  $\mathbb{Z}64$  = hi - lo + 1
  generate[[R extends Monoid[[R,  $\oplus$ ]], opr  $\oplus$ ]](body :  $\mathbb{Z}64 \rightarrow R$ ) : R =
    if size  $\leq$  max(b, 1) then
      r : R = Identity[[ $\oplus$ ]]
      i :  $\mathbb{Z}64$  = lo
      if i  $\leq$  hi then
        label done do
          while true do
            r := r  $\oplus$  body(i)
            if i  $\geq$  hi then exit done with () end
            i += 1
          end
        end
      end
    else
      mid =  $\lceil lo/2 \rceil$  +  $\lfloor hi/2 \rfloor$ 
      BlockedRange(lo, mid, b).generate(body) $\oplus$ 
      BlockedRange(mid + 1, hi, b).generate(body)
    end
end

```

This example generates the integers between *lo* and *hi* inclusive, with that ordering as the natural order of the generator. It does this using *recursive subdivision*. Recursive subdivision is the recommended technique for exposing large amounts of parallelism in a Fortress program because it adjusts easily to varying machine sizes and can be dynamically load balanced. In this example we divide the range in half if it is larger than the block size *b*; these two halves are computed in parallel (recall that the arguments to an operator are evaluated in parallel). If the range is smaller than *b*, then it is enumerated serially using a *while* loop, accumulating the result *r* as it goes.

The remainder of this section describes in detail the desugaring of generator lists and expressions with generators into invocations of the *generate* and *join* methods of the generators in the generator list. It then outlines how method overloading may be used to specialize the behavior of particular combinations of generators and reductions.

32.8.1 Simple Desugaring of Expressions with Generators

Each expression with generators is desugared into the following general form:

$$wrapper(\text{fn } () \Rightarrow \mathcal{T}[[R, \boxplus][gs]body)$$

where the desugaring must provide appropriate instantiations of *wrapper*, *body*, and the reduction static parameters, *R* and \boxplus . A simple and easily-understood desugaring “ $\mathcal{T}[[R, \boxplus][gs]body$ ” for generator lists is shown in Figure 32.1.

The desugaring of *GeneratorList* takes three parameters: a block of static parameters, *R* and \boxplus , the actual generator list (which we enclose in square brackets), and the *body* expression which should be used. Here and in subsequent desugarings, *v* in *v* \leftarrow *g* can stand either for a single variable or for a tuple of variables. We convert the

expr	type	wrapper	body	$\llbracket R, \boxplus \rrbracket$
$\sum_{gs} e$	R	\sum	e	$\llbracket R, + \rrbracket$
$lv := e, gs$	$()$	<i>noReduction</i>	$lv := e$	$\llbracket \text{NoReduction}, \oplus \rrbracket$
$\langle e \mid gs \rangle$	$\text{List}[E]$	<i>closeList</i>	<i>singletonOpen</i> (e)	$\llbracket \text{OpenList}, ++ \rrbracket$

Figure 32.2: Desugaring of expressions with generators. Top to bottom: big operators (here \sum is used as an example; the appropriate library function is called on the right-hand side), assignments, and comprehensions (here list comprehensions are shown; with the exception of array comprehensions, other comprehensions are similar to list comprehensions).

provided *GeneratorList* into a nested series of calls to *generate*. For example, when we perform the desugaring $T[\mathbb{Z}64, +][x \leftarrow xs, y \leftarrow ys, x \neq y](x \cdot y)$ we obtain the following code:

```
xs.generate[[Z64, +]](fn x =>
  ys.generate[[Z64, +]](fn y =>
    if x ≠ y then (x · y) else Identity[+] end))
```

Some example desugarings of expressions with generators are shown in Figure 32.2 for big operators, assignments, and list comprehensions (set and multiset comprehensions are similar to list comprehensions).

The simplest desugarings are the ones for big operators such as \sum . The type of the traversal corresponds to the type of the result. The body expression used is exactly the body expression of the big operator. The wrapper function is named by the big operator itself. For example, the \sum operator has the following declaration:

```
opr  $\sum$ [[ $R$  extends CommutativeMonoid[[ $R, +$ ]]]]( $rhs : () \rightarrow R$ ):  $R = rhs()$ 
```

Assignments desugar in a manner similar to big operators. However, they make use of the special *NoReduction* type, which is a singleton type which extends *CommutativeMonoid*[[*NoReduction*, \oplus]]. We can think of *NoReduction* as composing the writes of the assignments in parallel. The wrapper *noReduction* is defined as follows:

```
noReduction( $rhs : () \rightarrow \text{NoReduction}$ ):  $() = \text{do}$ 
   $rhs()$ 
   $()$ 
end
```

Lists also desugar in a similar way. The desugaring given in Figure 32.2 makes use of the *OpenList* type—such a list is constructed with an updatable tail cell, permitting partially-constructed lists to be appended in constant time using the $++$ (*DOUBLE.PLUS*) operator. The *closeList* operation converts the result into an ordinary non-updatable list.

An array comprehension simply desugars into a factory function call and a series of assignments:

<pre>[$i_1 = e_1 \mid gs_1$ $i_2 = e_2 \mid gs_2$... $i_n = e_n \mid gs_n$]</pre>	\longrightarrow	<pre>do $a = \text{array}()$ $a[i_1] := e_1, gs_1$ $a[i_2] := e_2, gs_2$... $a[i_n] := e_n, gs_n$ a end</pre>
---	-------------------	---

The desugaring of a *for* loop depends upon the set of reduction variables. We conceptually desugar the *for* loop with reduction variables, r_1, r_2, \dots, r_n , reduced using the reduction operator \oplus for type (T_1, T_2, \dots, T_n) as follows:

```
for  $gs$  do block end
 $\longrightarrow$ 
```

```

( $r_1, r_2, \dots, r_n$ )  $\oplus$  =  $\mathcal{T}[(T_1, T_2, \dots, T_n), \oplus][gs]$  (do
    ( $r_1, r_2, \dots, r_n$ ) : ( $T_1, T_2, \dots, T_n$ ) := Identity[ $\oplus$ ]
    block
    ( $r_1, r_2, \dots, r_n$ )
end)

```

In practice, a tuple type is not a monoid. If there is only one reduction variable, this is not a problem. If there are no reduction variables, we simply use the type `NoReduction` used in desugaring assignments. When there are multiple reduction variables, we use nested applications of types extending the trait `ReductionPair`. These types encode the common properties of the variables being reduced. Recall that every reduction variable must at least have type `Monoid`, so it is not difficult to guarantee that `ReductionPair` itself also extends `Monoid`.

32.8.2 Accounting for Dependencies among Generators

The naive desugaring for generator lists in Figure 32.1 assumes there are always data dependencies among generators. The actual desugaring makes use of the `join` method in the `Generator` trait to group together generators that have no data dependencies. The goal is to permit library code to define more efficient merged generators for generator pairs. For example, it is possible for the `join` method to take the generator list $i \leftarrow 1 \# 100, j \leftarrow 2 \# 200$ and generate a blocked two-dimensional traversal. This could then be joined with $k \leftarrow 3 \# 300$ to obtain a three-dimensional blocked traversal.

However, most generators will simply make use of the default definition of `join` which calls `SimplePairGenerator`:

```

object SimplePairGenerator[A extends Any, B extends Any]
  (outer : Generator[A], inner : Generator[B]) extends Generator[(A, B)]
  size : Z64 = outer.size * inner.size
  generate[R extends Monoid[R,  $\oplus$ ], opr  $\oplus$ ](body : (A, B)  $\rightarrow$  R) : R =
    outer.generate(fn (a : A)  $\Rightarrow$  inner.generate(fn (b : B)  $\Rightarrow$  body(a, b)))
  join[N extends Any](other : Generator[N]) : Generator[((A, B), N)] =
    SimpleMapGenerator(outer.join(inner.join(other)),
      (fn (a, (b, n))  $\Rightarrow$  ((a, b), n)))
end

```

Note how `SimplePairGenerator` itself overrides the `join` method. When we attempt to join an existing pair of joined generators, we first attempt to `join` the `inner` generator of the pair with the `other` generator (the new innermost generator) passed in. This means that every generator will have the opportunity to combine with both its left and right neighbors if neither has a dependency which prevents it. Note that we use a `SimpleMapGenerator`, which simply applies a function to the result of another generator, to re-nest the tuples produced by the nested `join` operation.

Which pairs of adjacent traversals are combined using `join`? This question is complicated by examples such as $i \leftarrow 1 : 100, j \leftarrow 1 : 100, k \leftarrow i : 100, l \leftarrow j : 100$. We can either combine i and j traversals, or we can combine j and k traversals. In the former case we can also combine k and l traversals. The Fortress compiler is free to choose any grouping subject to the following constraints:

- Two generators may not be combined using `join` if the second is data dependent upon the first.
- Generator order must be preserved when invoking `join`.
- When a chain of three or more generators is joined, the traversals must be combined left-associatively.

We can obtain a simple greedy desugaring which joins together traversals in accordance with the above rules by simply adding the following desugaring rule which takes precedence over those given in Figure 32.1 when each variable bound in v_1 does not occur free in g_2 .

$$\mathcal{T}[R, \boxplus][v_1 \leftarrow g_1, v_2 \leftarrow g_2, gs] \text{ body} = \mathcal{T}[R, \boxplus][(v_1, v_2) \leftarrow g_1.\text{join}(g_2), gs] \text{ body}$$

32.8.3 Using Overloading to Adapt Generators and Traversals

Overloaded instances of the *generate* method can be used to adapt a generator to the particular properties of the reduction being performed. For example, a commutative monoid need only maintain a single variable *result* containing the reduced value so far:

```
value object BlockedRange(lo: Z64, hi: Z64, b: Z64) extends Generator[Z64]
...
generate[R extends CommutativeMonoid[R, ⊕], opr ⊕](body : Z64 → R) : R = do
  result : R = Identity[⊕]
  traverse(l, u) =
    if u - l + 1 ≤ max(b, 1) then
      i : Z64 = l
      while i ≤ u do
        t = body(i)
        atomic result := result ⊕ t
        i += 1
      end
    ()
  else
    mid = ⌈l/2⌉ + ⌊u/2⌋
    (traverse(l, mid), traverse(mid + 1, u))
  ()
end
traverse(lo, hi)
result
end
end
```

The choice of whether to apply this transformation is left up to the author of the generator; when many iterations run in parallel the *result* variable becomes a scalability bottleneck and this technique should not be used.

Various other properties of the reduction operator can be exploited:

- Idempotent reductions permit redundant computation. For example, when computing the maximum element of a set it might be simpler to enumerate set elements more than once.
- On the other hand, sometimes a more efficient non-idempotent operator can be used for a reduction if the generator promises never to produce duplicates—this fact can be used to advantage in set, multiset, or map comprehensions.
- If the reduction operator has a zero, this can be used to exit early from a partial computation. This requires that the body expression have no visible side effects such as writes or `io` actions.

At the moment, the author of a `Generator` is responsible for taking advantage of opportunities such as these. In future, we expect some standardized support for efficient versions of various traversals based on experience with the definitions provided here.

32.8.4 Making a Serial Version of a Generator or Distribution

A generator *g* can be made sequential simply by calling the builtin function *sequential* as follows:

```
v ← sequential(g)
```

The resulting generator yields its elements in the natural order specified by the original generator. Several builtin generators (such as those for array indices) have an associated distribution. For these generators, *sequential* function simply re-distributes the underlying object as follows:

$$\mathit{sequential}(r) = \text{Sequential}.\mathit{distribute}(r)$$

As a convenient shorthand, the *sequential* function is also defined to work for distributions themselves. The complete signature for the overloads of *sequential* is as follows:

$$\begin{aligned} \mathit{sequential}[\![E \text{ extends Any}]\!](g : \text{Generator}[\![E]\!]) &: \text{Generator}[\![E]\!] \\ \mathit{sequential}[\![E \text{ extends Any}]\!](d : \text{Distribution}) &: \text{Distribution} \end{aligned}$$

The *sequential* function has special meaning to the Fortress implementation; there is no need to distinguish reduction variables in loops for which generator is surrounded by a direct call to *sequential*.

Note that at the moment there is no way to tell the compiler for performance reasons that we really mean it when we ask for sequentiality, as opposed to saying that we should preserve sequential semantics. Future versions of this specification may use the *Local* distribution for this purpose, or provide additional functions on generators which guarantee serial execution (rather than simply providing sequential semantics).

Chapter 33

Overloaded Functional Declarations

Fortress allows multiple functional declarations to be in scope of a particular program point. We call this overloading. Chapter 15 describes how to determine which overloaded declarations are applicable to a particular functional call, and when several are applicable, how to select the most specific one among them. In this chapter, we give a set of rules on overloaded declarations that guarantee there exists a most specific declaration for any given functional call. These rules are complicated by the presence of coercion, which may enlarge the set of declarations that are applicable to a functional call, as discussed in Chapter 17.

33.1 Principles of Overloading

Fortress allows multiple functional declarations of the same name to be declared in a single scope. However, recall from Chapter 7 the following shadowing rules:

- dotted method declarations shadow top-level function declarations with the same name, and
- dotted method declarations provided by a trait or object declaration or object expression shadow functional method declarations with the same name that are provided by a different trait or object declaration or object expression.

Also, note that a trait or object declaration or object expression must not have a functional method declaration and a dotted method declaration with the same name, either directly or by inheritance. Therefore, top-level functions can overload with other top-level functions and functional methods, dotted methods with other dotted methods, and functional methods with other functional methods and top-level functions.

Overloading functional declarations allows the benefits of polymorphic declarations. However, with these benefits comes the potential for ambiguous calls at run time. Fortress provides overloading rules for the *declarations* of functionals to eliminate the *possibility* of ambiguous calls at run time, whether or not these calls actually appear in the program.

Furthermore, these rules are checked statically. In fact, the overloading rules in Fortress allow the compiler to identify the statically most specific declaration for a particular call. Therefore an implementation strategy may be used in which the statically most specific declaration is identified statically, and the runtime dispatch mechanism need only consider dispatching among that declaration plus declarations that are more specific than that declaration (proof of this is given in Section B.2).

This chapter outlines several criteria for valid functional overloading. At any given program point, there is a set of overloaded declarations that are in scope. Fortress determines whether there is a possibility for ambiguous calls

from this set by comparing declarations pairwise. The following three sections each describes a rule to accept a pair of overloaded functional declarations. If a pair of overloaded declarations satisfies any one of the three rules, it is considered a valid overloading.

The overloading rules given in this chapter are disjoint. That is, at most one rule applies to each pair of overloaded declarations. It is possible for new rules to be added which allow additional overloadings.

Overloaded declarations must have static parameters that are identical (up to α -equivalence). Also, valid overloadings for declarations that contain keyword or varargs parameters are determined by analyzing the expansion of these declarations as described in Section 15.4. Therefore, static parameters, keyword parameters, and varargs parameters are ignored in the remainder of this chapter.

An operator method declaration whose name is one of the operator parameters (described in Section 11.5) of its enclosing trait or object may be overloaded with other operator declarations in the same component. Therefore, such an operator method declaration must satisfy the overloading rules with every operator declaration in the same component.

In addition, a functional which takes a single parameter of type `Any` (or a naked type parameter of bound `Any`) cannot be overloaded. This restriction prevents ambiguous overloadings such as the following:

```
makeSet[[T extends Any]](x: T): Set[[T]]
makeSet[[T extends Any]](x: T, y: T): Set[[T]]
```

The type parameter of `makeSet` may be instantiated with (\mathbb{Z}, \mathbb{Z}) or \mathbb{Z} . If this is the case then the call `makeSet(3, 4)`, where 3 and 4 have type \mathbb{Z} , is ambiguous.

Section 33.2 states the *Subtype Rule*, which stipulates that the parameter type of one declaration be a subtype of the parameter type of the other. In this case, there is no possibility of ambiguous calls, because one declaration is more specific than the other. This section also places a restriction on the return types of the overloaded declarations to ensure that type safety is not violated. Section 33.3 defines the *Incompatibility Rule* that, if satisfied by a pair of declarations, guarantees that neither declaration is applicable to the same functional call. In Section 33.4, the *Meet Rule* requires the existence of a declaration that is more specific than both overloaded declarations in the situation that both are applicable to a given call.

In the remainder of this chapter we build on the terminology and notation defined in Chapter 15 and Chapter 17.

33.2 Subtype Rule

If the parameter type of one declaration is a subtype of the parameter type of another (and they are not the same) then there is no ambiguity between these two declarations: for every call to which both are applicable, the first is more specific. This is the basis of the Subtype Rule.

The Subtype Rule also requires a relationship between the return types of the two declarations. Without such a requirement, a program may violate type safety.

The Subtype Rule for Functions and Functional Methods: Suppose that $f(P) : U$ and $f(Q) : V$ are two distinct function or functional method declarations in a single scope. If $P \prec Q$ and $U \preceq V$ then $f(P)$ and $f(Q)$ are a valid overloading.

The Subtype Rule for Dotted Methods: Suppose that $P_0.f(P) : U$ and $Q_0.f(Q) : V$ are two distinct dotted method declarations provided by a trait or object C . If $(P_0, P) \prec (Q_0, Q)$ and $U \preceq V$ then $P_0.f(P)$ and $Q_0.f(Q)$ are a valid overloading.

33.3 Incompatibility Rule

The basic idea behind the Incompatibility Rule is that if there is no call to which two overloaded declarations are both applicable then there is no potential for ambiguous calls. In such a case, we say that the declarations are incompatible. Without coercion, incompatibility is equivalent to exclusion. However, the presence of coercion complicates the definition of incompatibility. To formally define incompatibility we first define the following notation. For types T and U , we say that T and U *do not share coercions*, and write $T \not\sim U$, if any type that coerces to T excludes any type that coerces to U :

$$T \not\sim U \iff \forall A, B : A \rightarrow T \wedge B \rightarrow U \implies A \diamond B.$$

We say that T is *incompatible with* U , and write $T \blacklozenge U$, if T and U exclude, reject each other, and do not share coercions:

$$\begin{aligned} T \blacklozenge U &\iff T \diamond U \wedge T \rightarrow U \wedge U \rightarrow T \wedge T \not\sim U \\ &\iff T \diamond U \\ &\quad \wedge (\forall A : A \rightarrow T \implies A \diamond U) \\ &\quad \wedge (\forall B : B \rightarrow U \implies B \diamond T) \\ &\quad \wedge (\forall A, B : A \rightarrow T \wedge B \rightarrow U \implies A \diamond B) \end{aligned}$$

Note that if $T \blacklozenge U$ then no type is substitutable for both T and U .

The Incompatibility Rule for Functions and Functional Methods: Suppose that $f(P)$ and $f(Q)$ are two distinct function or functional method declarations in a single scope. If $P \blacklozenge Q$ then $f(P)$ and $f(Q)$ are a valid overloading.

The Incompatibility Rule for Dotted Methods: Suppose that $P_0.f(P)$ and $Q_0.f(Q)$ are two distinct dotted method declarations provided by a trait or object C . If $P \blacklozenge Q$ then $P_0.f(P)$ and $Q_0.f(Q)$ are a valid overloading.

33.4 Meet Rule

If neither the Subtype Rule nor the Incompatibility Rule holds for a pair of overloaded declarations then the declarations introduce the possibility of ambiguity. To avoid this ambiguity, we require a *disambiguating declaration*; this is, for every call to which both declarations are applicable, there must be a third, more specific, declaration that is also applicable. Thus, at run time, neither of the pair of declarations is executed because the disambiguating declaration is also applicable, and it is more specific than both.

The Meet Rule for Functions: Suppose that $f(P)$ and $f(Q)$ are two function declarations in a single scope such that neither P nor Q is a subtype of the other and P and Q are not incompatible with one another. Let \mathcal{S} be the set of types that P defines coercions from and \mathcal{T} be the set of types that Q defines coercions from. $f(P)$ and $f(Q)$ are a valid overloading if all of the following hold:

- either $P \diamond Q$ or there is a declaration $f(P \cap Q)$ in the scope, and
- either $P \triangleleft Q$ or $Q \triangleleft P$ or for all $P' \in \mathcal{S}$ and $Q' \in \mathcal{T}$ one of the two conditions holds:
 - $P' \diamond Q'$, or
 - there is a declaration $f(P' \cap Q')$ in the scope.

Recall that $P \cap Q$ is the intersection of types P and Q as defined in Section 8.8. If for some type S we have $S \preceq P$ and $S \preceq Q$ then $S \preceq (P \cap Q)$, but it's not necessarily the case that $S = (P \cap Q)$ since another type may be more specific than both P and Q . For example, suppose the following:

```

trait S comprises {U, V} end
trait T comprises {V, W} end
trait U extends S excludes W end
trait V extends {S, T} end
trait W extends T end

f(s: S) = 1
f(t: T) = 1
f(v: V) = 1

```

Because of the `comprises` clauses of S and T and the `excludes` clause of U , any subtype of both S and T must be a subtype of V . Thus, $V = S \cap T$, and the declaration $f(V)$ “disambiguates” $f(S)$ and $f(T)$, i.e., it is applicable to and more specific for any call to which both $f(S)$ and $f(T)$ are applicable.

The Meet Rule should not be difficult to obey, especially because the compiler can give useful feedback. For example:

```

foo(x: Number, y: Z64) = ...
foo(x: Z64, y: Number) = ...

```

Assuming that $Z64 \prec \text{Number}$, the compiler reports that these two declarations are a problem because of ambiguity and suggests that a new declaration for $foo(Z64, Z64)$ would resolve the ambiguity. As another example, consider:

```

bar(x: Printable) = ...
bar(x: Throwable) = ...

```

Assuming that `Printable` and `Throwable` are neither comparable by the subtyping relation nor disjoint, the compiler reports that these two declarations are a problem because `Printable` and `Throwable` are incomparable but possibly overlapping types. As a result, these two declarations are statically rejected.

Unlike for functions, the Meet Rule for dotted methods only applies to dotted methods that are provided by the same trait or object. This is possible because two dotted methods are applicable to a given call $A_0.f(A)$ only if they are both provided by the trait or object A_0 .

The Meet Rule for Dotted Methods: Suppose that $P_0.f(P)$ and $Q_0.f(Q)$ are two dotted method declarations provided by a trait or object C such that neither (P_0, P) nor (Q_0, Q) is a subtype of the other and P and Q are not incompatible with one another. Let \mathcal{S} be the set of types that P defines coercions from and \mathcal{T} be the set of types that Q defines coercions from. $P_0.f(P)$ and $Q_0.f(Q)$ are a valid overloading if all of the following hold:

- either $P \diamond Q$ or there is a declaration $R_0.f(P \cap Q)$ provided by C with $R_0 \preceq (P_0 \cap Q_0)$, and
- either $P \triangleleft Q$ or $Q \triangleleft P$ or for all $P' \in \mathcal{S}$ and $Q' \in \mathcal{T}$ one of the two conditions holds:
 - $P' \diamond Q'$, or
 - there is a declaration $R_0.f(P' \cap Q')$ provided by C with $R_0 \preceq (P_0 \cap Q_0)$.

Recall that functional methods can be viewed semantically as top-level functions, as described in Section 9.2. However, treating functional methods as top-level functions for determining valid overloading is too restrictive. In the following example:

```

trait Z
  opr -(self): Z

```

```

end
trait  $\mathbb{R}$ 
  opr  $-(self): \mathbb{R}$ 
end

```

if the functional methods were interpreted as top-level functions then this program would define two top-level functions with parameter types \mathbb{Z} and \mathbb{R} . These declarations would be statically rejected as an invalid overloading because there is no relation between \mathbb{Z} and \mathbb{R} ; another trait may extend them both without declaring its own version of the functional method which may lead to an ambiguous call at run time. However, notice that declarations are ambiguous only for calls on arguments that extend both \mathbb{Z} and \mathbb{R} , and any type that extends both can include a new declaration that disambiguates them. We use this intuition to allow such overloadings.

The Meet Rule for Functional Methods: Suppose that $f(P)$ and $f(Q)$ are two functional method declarations occurring in trait or object declarations or object expressions such that neither P nor Q is a subtype of the other and P and Q are not incompatible with one another. Let $f(P)$ and $f(Q)$ have self parameters at i and j respectively. Also, let \mathcal{S} be the set of types that P defines coercions from and \mathcal{T} be the set of types that Q defines coercions from. $f(P)$ and $f(Q)$ are a valid overloading if all of the following hold:

- $i = j$
- either $P \diamond Q$ or if there exists a trait or object C that provides both $f(P)$ and $f(Q)$ then $P \neq Q$ and there is a declaration $f(P \cap Q)$ provided by C , and
- either $P \triangleleft Q$ or $Q \triangleleft P$ or for all $P' \in \mathcal{S}$ and $Q' \in \mathcal{T}$ one of the two conditions holds:
 - $P' \diamond Q'$, or
 - there is a declaration $f(P' \cap Q')$ provided by C .

Notice that the Meet Rule for functional methods requires the self parameters of two overloaded declarations to be in the same position. This requirement guarantees that no ambiguity is caused by the position of the self parameter. Two declarations which differ in the position of the self parameter must satisfy either the Subtype Rule or the Incompatibility Rule to be a valid overloading.

Functional method declarations can overload with function declarations. A valid overloading between a functional method declaration and a function declaration is determined by applying the (more restrictive) Meet Rule for functions to both declarations.

33.5 Coercion and Overloading Resolution

The overloading rules given in this chapter are sufficient to prove the following two facts:

1. If no declaration is applicable to a static call but there is a declaration that is applicable with coercion then there exists a single most specific declaration that is applicable with coercion to the static call.
2. If any declaration is applicable to a static call then there exists a single most specific declaration that is applicable to the static call and a single most specific declaration that is applicable to the corresponding dynamic call.

Moreover, we can prove that the most specific declaration that is applicable to a dynamic call is more specific than the most specific declaration that is applicable to the corresponding static call.

Appendix B formally proves that the rules discussed in the previous sections guarantee the static resolution of coercion (described in Section 17.5) is well defined for functions (the case for methods is analogous). Also in Appendix B is a proof that the overloading rules for function declarations are sufficient to guarantee no undefined nor ambiguous calls at run time (again, the case for methods is analogous).

Chapter 34

Operator Declarations

An operator declaration may appear anywhere a top-level function or method declaration may appear. Operator declarations are like other function or method declarations in all respects except that an operator declaration has `opr` and has an operator name (see Section 16.1 for a discussion of valid operator names) instead of an identifier. The precise placement of the operator name within the declaration depends on the fixity of the operator. Like other functionals, operators may have overloaded declarations (see Chapter 15 for a discussion of overloading). These overloads may be of the same or differing fixities.

Syntax:

```
OpDecl ::= FnDecl
FnDecl ::= FnDef
           | AbsFnDecl
FnDef ::= FnMods? FnHeaderFront FnHeaderClause = Expr
FnHeaderFront ::= OpHeaderFront
OpHeaderFront ::= opr StaticParams? (LeftEncloser | Encloser) Params (RightEncloser | Encloser)
                ( := ( SubscriptAssignParam ) )?
                | opr StaticParams? ValParam Op
                | opr (Op | Encloser) StaticParams? ValParam
SubscriptAssignParam ::= Varargs
                       | Param
```

An operator declaration has one of seven forms: infix/multifix operator declaration, prefix operator declaration, postfix operator declaration, nofix operator declaration, bracketing operator declaration, subscripting operator method declaration, and subscripted assignment operator method declaration. Each is invoked according to specific rules of syntax. An operator method declaration should be a functional method declaration, a subscripting operator method declaration, or a subscripted assignment operator method declaration.

34.1 Infix/Multifix Operator Declarations

An infix/multifix operator declaration has `opr` and then an operator name where a functional declaration would have an identifier. The declaration must not have any keyword parameters, and must be capable of accepting at least two arguments. It is permissible to use a `varargs` parameter; in fact, this is a good way to define a multifix operator. Static parameters (described in Chapter 11) may also be present, between the operator and the parameter list.

An expression consisting of an infix operator applied to an expression will invoke an infix/multifix operator declaration. The compiler considers all infix/multifix operator declarations for that operator that are both accessible and applicable,

and the most specific operator declaration is chosen according to the usual rules for overloaded functionals. If the expression is actually multifix, the invocation will pass more than two arguments.

An infix/multifix operator declaration may also be invoked by a prefix or nofix (but not a postfix) operator application if the declaration is applicable.

Note that superscripting (\wedge) may be defined using an infix operator declaration even though it has very high precedence and cannot be used as a multifix operator. (An operator declaration for superscripting should have exactly two value parameters.)

Example:

```
opr MAX[[T extends Rational]](x:T, y:T):T = if x > y then x else y end
```

34.2 Prefix Operator Declarations

A prefix operator declaration has `opr` and then an operator name where a functional declaration would have an identifier. The declaration must have one value parameter, which must not be a keyword parameter or varargs parameter. Static parameters may also be present, between the operator and the parameter list.

An expression consisting of a prefix operator applied to an expression will invoke a prefix operator declaration. The compiler considers all prefix and infix/multifix operator declarations for that operator that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals.

Example:

```
opr ~(x:Widget):Widget = x.invert()
```

34.3 Postfix Operator Declarations

A postfix operator declaration has `opr` where a functional declaration would have an identifier; the operator name itself *follows* the parameter list. The declaration must have one value parameter, which must not be a keyword parameter or varargs parameter. Static parameters may also be present, between `opr` and the parameter list.

An expression consisting of a postfix operator applied to an expression will invoke a postfix operator declaration. The compiler considers all postfix operator declarations for that operator that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals.

Example:

```
opr (n:Integer)! = [[i ← 1:n]i (* factorial *)
```

34.4 Nofix Operator Declarations

A nofix operator declaration has `opr` and then an operator name where a functional declaration would have an identifier. The declaration must have no parameters.

An expression consisting only of a nofix operator will invoke a nofix operator declaration. The compiler considers all nofix and infix/multifix operator declarations for that operator that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals.

Uses for nofix operators are rare, but those rare examples are very useful. For example, the colon operator is used to construct subscripting ranges, and it is the nofix declaration of `:` that allows a lone `:` to be used as a subscript:

`opr` : () = ImplicitRange

34.5 Bracketing Operator Declarations

A bracketing operator declaration has `opr` where a functional declaration would have an identifier. The value parameter list, rather than being surrounded by parentheses, is surrounded by the brackets being defined. A bracketing operator declaration may have any number of parameters, keyword parameters, and varargs parameters in the value parameter list. Static parameters may also be present, between `opr` and the parameter list. Any paired Unicode brackets may be so defined *except* ordinary parentheses and white square brackets.

An expression consisting of zero or more comma-separated expressions surrounded by a bracket pair will invoke a bracketing operator declaration. The compiler considers all bracketing operator declarations for that type of bracket pair that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals. For example, the expression $\langle p, q \rangle$ might invoke the following bracketing method declaration:

```
(* angle bracket notation for inner product *)
opr ⟨x : Vector, y : Vector⟩ = ∑[i ← x.indices] xi · yi

(* vector space norm (may not be the most efficient) *)
opr ‖x : Vector‖ = sqrt⟨x, x⟩
```

34.6 Subscripting Operator Method Declarations

A subscripting operator method declaration has `opr` where a method declaration would have an identifier. The value parameter list, rather than being surrounded by parentheses, is surrounded by a pair of brackets. A subscripting operator method declaration may have any number of value parameters, keyword parameters, and varargs parameters in that value parameter list. Static parameters may also be present, between `opr` and the parameter list. Any paired Unicode brackets may be so defined *except* ordinary parentheses and white square brackets; in particular, the square brackets ordinarily used for indexing may be used.

An expression consisting of a subexpression immediately followed (with no intervening whitespace) by zero or more comma-separated expressions surrounded by brackets will invoke a subscripting operator method declaration. Methods for the expression preceding the bracketed expression list are considered. The compiler considers all subscripting operator method declarations that are both accessible and applicable, and the most specific method declaration is chosen according to the usual overloading rules. For example, the expression foo_p might invoke the following subscripting method declaration because expressions in the square brackets are rendered as subscripts:

```
(* subscripting method *)
opr [x : BizarroIndex] = self.bizarroFetch(x)
```

34.7 Subscripted Assignment Operator Method Declarations

A subscripted assignment operator method declaration has `opr` where a method declaration would have an identifier. The value parameter list, rather than being surrounded by parentheses, is surrounded by a pair of brackets; this is then followed by the operator `:=` and then a second value parameter list in parentheses, which must contain exactly one non-keyword value parameter. A subscripted assignment operator method declaration may have any number of value parameters within the brackets; these value parameters may include keyword parameters and varargs parameters. A result type may appear after the second value parameter list, but it must be (). Static parameters may also be present,

between `opr` and the first parameter list. Any paired Unicode brackets may be so defined *except* ordinary parentheses and white square brackets; in particular, the square brackets ordinarily used for indexing may be used.

An assignment expression consisting of an expression immediately followed (with no intervening whitespace) by zero or more comma-separated expressions surrounded by brackets, followed by the assignment operator `:=`, followed by another expression, will invoke a subscripted assignment operator method declaration. Methods for the expression preceding the bracketed expression list are considered. The compiler considers all subscript operator method declarations that are both accessible and applicable, and the most specific method declaration is chosen according to the usual overloading rules. When a compound assignment operator (described in Section 13.8) is used with a subscripting operator and a subscripted assignment operator, for example `a3 += k`, both a subscripting operator declaration and a subscripted assignment operator declaration are required. For example, the assignment `foop := myWidget` might invoke the following subscripted assignment method declaration:

```
(* subscripted assignment method *)
opr [x : BizarroIndex] := (new Value : Widget) = self.bizarroInstall(x, new Value)
```

34.8 Conditional Operator Declarations

A *conditional operator* is a binary operator (other than `:'`) that is immediately followed by `:'`; see Section 16.6. A conditional operator expression `x@:y` is syntactic sugar for `x@(fn () => y)`; that is, the right-hand operand is converted to a “thunk” (zero-parameter function) that then becomes the right-hand operand of the corresponding unconditional operator. Therefore a conditional operator is simply implemented as an overloading of the operator that accepts a thunk.

It is also permitted for a conditional operator to have a preceding as well as a following colon. A conditional operator expression `x:@:y` is syntactic sugar for `(fn () => x)@(fn () => y)`; that is, each operand is converted to a thunk. This mechanism is used, for example, to define the results-comparison operator `:~:`, which takes exceptions into account.

The conditional \wedge and \vee operators for boolean values, for example, are implemented as methods in this manner:

```
opr  $\wedge$ (self, other: Boolean)      = if self then other else false end
opr  $\wedge$ (self, other: () → Boolean) = if self then other() else false end
opr  $\vee$ (self, other: Boolean)      = if self then true else other end
opr  $\vee$ (self, other: () → Boolean) = if self then true else other() end
```

34.9 Big Operator Declarations

A *big operator* such as \sum or \prod is declared as a usual operator declaration. See Section 32.8.1 for an example declaration of a big operator. A big operator application is called a *reduction expression* and described in Section 13.17.

Chapter 35

Dimensions and Units Declarations

Syntax:

```
DimUnitDecl ::= dim Id (= DimRef)? (default Id)?  
                | (unit | SI_unit) Id+ (: DimRef)? (= Expr)?  
                | dim Id (= DimRef)? (unit | SI_unit) Id+ (= Expr)?
```

35.1 Dimensions Declarations

Dimensions may be explicitly declared; every declared dimension must be declared at the top level of a program component, not within a block expression or trait. Other dimensions may be constructed by multiplying and dividing other dimensions, as described in Chapter 18. An explicitly declared dimension may be a *base dimension* (with no definition specified) or a *derived dimension* (with a definition specified in the form of an initialization expression).

The set of all dimensions has the algebraic structure of a free abelian group. The identity element of this group is the dimension `Unity`, which represents dimensionlessness.

For every two dimensions D and E , there is a dimension DE (which may also be written $D \cdot E$), corresponding to the product of the dimensions D and E and a dimension D/E , corresponding to the quotient of the dimensions D and E . The syntactic sugar $1/D$ is equivalent to `Unity/D` for all dimensions D . A dimension can be raised to a rational power where both the numerator and the denominator of the rational power must be a valid `nat` *parameter* instantiation (as described in Section 11.2); D^0 is the same as `Unity`, D^1 is the same as D , and D^{m+n} is the same as $D^m D^n$. The syntactic sugar D^{-n} is the same as `Unity/Dn`.

Here are some examples of base dimension declarations:

```
dim Length  
dim Mass  
dim Time  
dim ElectricCurrent
```

Here are some examples of computed dimensions:

```
Length/Time  
Velocity/Time  
Length · Mass/Time2  
Length Mass Time-2  
ElectricCurrent/Length2
```

and here some of these computed dimensions are given names through the use of derived dimension declarations:

```
dim Velocity = Length/Time
dim Acceleration = Velocity/Time
dim CurrentDensity = ElectricCurrent/Length2
```

35.2 Units Declarations

Every unit belongs to exactly one dimension, which is the type of the unit. A dimension may have more than one unit, but one of these units may be singled out as the *default unit* for that dimension by adding a `default` clause:

```
dim Length default meter
dim Mass default kilogram
dim Time default second
```

The default unit is used when a numerical type is multiplied by a dimension to produce a new type (see Chapter 18). If no default clause is specified for a base dimension, then it has no default unit. If no default clause is specified for a derived dimension, then it has a default unit if and only if all the dimensions mentioned in its initialization expression have defaults, in which case its default unit is calculated using the initialization expression with each dimension replaced by its default unit.

Some units are explicitly declared; every declared unit must be declared at the top level of a program component, not within a block expression or trait. Other units may be constructed by multiplying and dividing other units. An explicitly declared unit may be a *base unit* (with no definition specified) or a *derived unit* (with a definition specified in the form of an initialization expression).

The set of all units, like the set of all dimensions, has the algebraic structure of a free abelian group. The identity element of this group is the unit dimensionless, of dimension `Unity`. Note that there may be other units of dimension `Unity`, such as radian and steradian, but only dimensionless is the identity of the group of all units. (Note that there is a straightforward homomorphism of units onto dimensions, wherein every unit is mapped to its dimension.)

Here are some examples of base unit declarations:

```
unit meter : Length
unit kilogram : Mass
unit second : Time
unit ampere : ElectricCurrent
```

Here are some examples of computed units:

```
meter/second
(meter/second)/second
meter · kilogram/second2
meter kilogram second-2
ampere/meter2
```

and here some computed units are given names through the use of derived unit declarations:

```
unit newton: Force = meter · kilogram/second2
unit joule: Energy = newton meter
unit pascal: Pressure = newton/meter2
```

In the preceding examples, the initialization expression for each unit is itself a unit. It is also permitted for the initialization expression to be a dimensioned numerical value, in which case the unit being declared is related to the unit of the dimensioned numerical value by a numerical conversion factor.

As with an ordinary variable declaration, one may omit the dimension for a unit if there is an initialization expression; the dimension of the declared unit is the dimension of the unit of the expression.

Every unit can be reduced to a canonical value as follows. A base unit is multiplied by the value 1; a `unit` parameter is multiplied by the value 1; a defined unit is replaced by its initialization expression and then every unit in that expression is replaced by its canonical form; and finally all arithmetic is performed so as to reduce the units to a single unit and the numerical values to a single numerical value. A dimensioned value with unit U is convertible by the `in` operator to a value with unit V if the canonical values for U and V have the same unit; the conversion involves multiplying the numerical value by the ratio of the numerical value of the canonical form of V to the numerical value of the canonical form of U .

For example, given the declarations:

```
dim Length
unit meter: Length; unit meters = meter
unit kilometer: Length = 103meter; unit kilometers = kilometer
unit inch: Length = 2.54 × 10-2meter; unit inches = inch
unit foot: Length = 12 inch; unit feet = foot
unit mile: Length = 5280 foot; unit miles = mile
```

then one can say `3 miles in kilometers` and the `in` operator will multiply the numerical value 3 by the amount of $((2.54 \times 10^{-2})(12)(5280)/10^3)$, or 25146/15625.

Notice the subtle difference between these two declarations:

```
unit radian = meter/meter
unit radian = 1 meter/meter
```

The first declaration defines `radian` to be equivalent to dimensionless, and so a value with unit `radian` can be used anywhere a dimensionless value can be used, and vice versa. The second declaration defines `radian` to be convertible to dimensionless but not equivalent, and so it is necessary to use the `in` operator (or multiplication and division by `radian`) to convert between values in radians and truly dimensionless values.

35.3 Abbreviating Dimension and Unit Declarations

For convenience, three forms of syntactic sugar are provided when declaring dimensions and units. First, in a `unit` declaration one may mention more than one name before the colon, and the extra names are defined to be synonyms for the first name; thus

```
unit foot feet ft: Length
```

means exactly the same thing as

```
unit foot: Length
unit feet: Length = foot
unit ft: Length = foot
```

Second, instead of the reserved word `unit` one may use the reserved word `SI_unit`, which has the effect of defining not only the specified names but also names with the various SI prefixes attached. If more than one name is specified, then the last name is assumed to be a symbol and has symbol prefixes (such as `M` and `n`) attached; all other names have the full prefixes (such as `mega` and `nano`) attached. Thus

```
SI_unit name1 name2 name3:...
```

may be regarded as an abbreviation for

```

unit name1 name2 name3:...
unit yottaname1 yottaname2 Yname3 = 1024name1
unit zettaname1 zettaname2 Zname3 = 1021name1
unit exaname1 exaname2 Ename3 = 1018name1
unit petaname1 petaname2 Pname3 = 1015name1
unit teraname1 teraname2 Tname3 = 1012name1
unit giganame1 giganame2 Gname3 = 109name1
unit meganame1 meganame2 Mname3 = 106name1
unit kiloname1 kiloname2 kname3 = 103name1
unit hectoname1 hectoname2 hname3 = 102name1
unit dekaname1 dekaname2 daname3 = 10name1
unit deciname1 deciname2 dname3 = 10-1name1
unit centiname1 centiname2 cname3 = 10-2name1
unit milliname1 milliname2 mname3 = 10-3name1
unit microname1 microname2 μname3 = 10-6name1
unit nanoname1 nanoname2 nname3 = 10-9name1
unit piconame1 piconame2 pname3 = 10-12name1
unit femtoname1 femtoname2 fname3 = 10-15name1
unit attoname1 attoname2 aname3 = 10-18name1
unit zeptoname1 zeptoname2 zname3 = 10-21name1
unit yoctoname1 yoctoname2 yname3 = 10-24name1

```

where μ is the Unicode character U+00B5 MICRO SIGN. Third, a dim declaration and a unit or SI_unit declaration may be collapsed into a single declaration by writing the unit or SI_unit declaration in place of the default clause in the dim declaration and omitting the colon and dimension from the unit declaration. Thus

```

dim Length SI_unit meter meters m
dim Power = Energy/Time SI_unit watt watts W = joule/second

```

is understood to abbreviate

```

dim Length default meter; SI_unit meter meters m: Length
dim Power = Energy/Time default watt; SI_unit watt watts W: Power = joule/second

```

In this way the names of the seven SI base units, along with all possible plural and prefixed forms, may be concisely defined as follows:

```

dim Length SI_unit meter meters m
dim Mass default kilogram; SI_unit gram grams g: Mass
dim Time SI_unit second seconds s
dim ElectricCurrent SI_unit ampere amperes A
dim Temperature SI_unit kelvin kelvins K
dim AmountOfSubstance SI_unit mole moles mol
dim LuminousIntensity SI_unit candela candelas cd

```

Note the subtle difference in the declaration of Mass that allows the default unit to be kilogram rather than gram.

35.4 Absorbing Units

Syntax:

```

StaticParam ::= Id Extends? (absorbs unit)?
              | unit Id (: DimRef)? (absorbs unit)?

```

The declaration of a type parameter or a `unit` parameter for a parameterized trait may contain a clause “`absorbs unit`”; at most one static parameter of a trait may have this clause. An instance of a parameterized trait with a static parameter that “`absorbs unit`” may be multiplied or divided by a unit, the result being another instance of that parameterized trait in which the static argument corresponding to the unit-absorbing parameter has been multiplied or divided by the unit.

A few examples should make this clear. Given the declaration

```
trait Vector[EltType extends Number absorbs unit, nat len] ... end
```

then `Vector[[Float, 3]]meter` means the same as `Vector[[Float meter, 3]]`, and `Vector[[Float, 3]]/second` means the same as `Vector[[Float/second, 3]]`. Therefore, `[(3 m)(2 m)(5 m)]` means the same as `[3 2 5]m`. Similarly, given the declaration

```
trait Float[unit U absorbs unit, nat e, nat s] ... end
```

then `Float[[meter, 11, 53]]/second` means the same as `Float[[meter/second, 11, 53]]`, and

```
Float[[dimensionless, 8, 24]]meter kilogram/second2
```

means the same as

```
Float[[dimensionless meter kilogram/second2, 8, 24]],
```

which is the same as

```
Float[[meter kilogram/second2, 8, 24]].
```

This is the mechanism by which meaning is given to the multiplication and division of library-defined types by units.

Chapter 36

Support for Domain-Specific Languages

To support syntax for domain-specific languages, and to allow the Fortress language to grow with time, programmers can extend the basic syntax of Fortress in their programs. Such extensions are possible through the use of *syntax expanders*. Syntax expanders must be defined in the top level of a simple component.

36.1 Definitions of Syntax Expanders

Syntax:

```
ExternalSyntax ::= syntax OpenExpander Id CloseExpander = Expr  
OpenExpander ::= Id  
                | LeftEncloser  
CloseExpander ::= Id  
                | RightEncloser  
                | end
```

The definition of a syntax expander starts with *syntax*, followed by an *opening delimiter*, followed by a *contents parameter*, followed by a *terminating delimiter*, followed by an =, and a subexpression. The opening delimiter must be either an identifier or the opening member of an enclosing operator. The contents parameter must be an identifier (see Section 5.15). The terminating delimiter must be either an identifier, the terminating member of an enclosing operator, or *end*. If either the opening delimiter or the closing delimiter is *part* of an enclosing operator, the opening and closing delimiters must both be parts of enclosing operators, and they must match, or it is a static error. Because delimiters conceptually delimit blocks, just as *do* and *end* delimit blocks, delimiters of syntax expanders are rendered as reserved words. The subexpression of a syntax expander has type `Fortress.Ast.SyntaxTree`. This `SyntaxTree` must be that of a Fortress expression. Here is an example:

```
syntax sql e end = parseSQL(e)
```

where *parseSQL* is a function that takes an argument of type `Fortress.Lang.SourceAssembly` (a sequence of Unicode characters and abstract syntax trees), interprets it as an SQL query, and returns an expression with type `SyntaxTree` consisting of constructor calls to SQL syntax nodes (defined in some SQL library).

At a use site, all characters between the opening delimiter and the terminating delimiter are turned into a `SourceAssembly` (see Section 36.4 for a more detailed description of how this conversion is achieved). The resulting `SourceAssembly` is bound to the contents parameter of the syntax expander. The use site is then expanded by evaluating the body of the expander. Every use site of a syntax expander must occur in an expression context, or it is a static error.

For example, we could define *parseSQL* so that a use site such as:

```
sql
  SELECT spectral_class FROM stars
end
```

would be expanded into the following Fortress SyntaxTree:

```
Call(Empty,
  List(VarRef(Identifier("SqlQuery")),
    Call(Empty,
      List(VarRef(Identifier("Select")),
        String("spectral_class")),
      Call(Empty,
        List(VarRef(Identifier("From")),
          String("stars"))))))
```

(The Empty lists passed to Calls are the lists of static parameters to these calls). Note that this SyntaxTree corresponds to the following Fortress concrete syntax:

```
SqlQuery(Select("spectral_class"), From("stars"))
```

36.2 Declarations of Syntax Expanders

Syntax:

```
AbsExternalSyntax ::= syntax OpenExpander Id CloseExpander
```

A *declaration* of a syntax expander is syntactically identical to the definition of a syntax expander, except that = and the body of the expander (i.e., the expression following the = sign in an expander definition) are elided. Syntax expander declarations must occur only in APIs. A component that exports an API *A* must provide, for each syntax expander declaration *d* in *A*, a syntax expander definition with a header identical to *d*.

36.3 Restrictions on Delimiters

Consider the set *S* of syntax expander declarations imported by a component or API *B*, along with the syntax expanders defined or declared in *B* directly. Every expander in *S* must have a distinct opening delimiter, or it is a static error. Moreover, the terminating delimiter of each syntax expander must be distinct from every opening delimiter of every syntax expander in *S*, or it is a static error.

36.4 Processing Syntax Expanders

In a given component, only syntax expander declarations appearing in APIs imported by the component may be used. A component exporting an API *A* that includes a syntax expander must not import APIs that are not also imported by *A*, or it is a static error. This restriction ensures that all names in scope of the definition of the syntax expander are also in scope of any component importing the syntax expander in *A*. Additionally, all use sites of a syntax expander must occur in simple components other than the defining component. Furthermore, multiple components containing syntax expanders must not be cyclically linked. These restrictions avoid pathologies with nontermination during expansion.

Finally, to maintain proper separation of test code, a syntax expander definition is statically forbidden from referring to variables or functions that have modifier `test`.

Because syntax expanders are defined at the top level of program components, and because they are syntactically distinguished, they can be identified before scanning or parsing (but after ASCII conversion). Use sites are then identified and expanded before parsing occurs.

Syntax expansion takes a sequence of Unicode characters and yields a sequence of Unicode characters and syntax trees, where all the syntax expanders have been replaced by syntax trees. This result has type `SourceAssembly`. Syntax expansion proceeds from left to right as follows. First, the source is scanned for opening delimiters of syntax expanders, stopping at the leftmost one. We call this scanning *Fortress-source scanning*.

If the opening delimiter of a syntax expander is encountered during Fortress-source scanning, the source is scanned rightward until the first occurrence of either an opening delimiter of some syntax expander (possibly another use of the same expander), or the terminating delimiter of that syntax expander, is found. (If no matching terminating delimiter is found in the remainder of the program, it is a static error.) We call this scanning *expander scanning*.

If an opening delimiter is encountered before the terminating delimiter, there is a *nested use site* of another syntax expander. The nested use site is processed, and then expander scanning continues rightward of that use site. Thus, the processing of syntax expanders is recursive, syntax expanders may be nested arbitrarily, and expanders are processed from the leftmost-innermost occurrence outward and rightward.

When the terminating delimiter is encountered during expander scanning, the scanning is terminated, the resulting `SourceAssembly` is bound to the contents parameter of the expander, and then the body of the expander is evaluated. The result of evaluating this body has type `SyntaxTree`, and it is placed into the resulting `SourceAssembly` in place of the expander in the scanned source.

36.4.1 Introduced Variable Names

Often, when expanding concrete syntax for a domain-specific language, it is useful to introduce variable binding constructs into the resulting `SyntaxTree`. It is required that such bindings, in general, respect the rules of hygiene and referential transparency [7]. Several aspects of the Fortress semantics allow the library programmer to ensure referential transparency of syntax expanders:

1. A component exporting an API *A* that includes a syntax expander must not import APIs that are not also imported by *A*.
2. Syntax expanders must expand to `SyntaxTrees` of expressions.
3. Shadowing of identifiers is not allowed.

Thus, syntax expanders cannot expand into new top-level identifiers. Moreover, provided that the library programmer is careful to ensure that all top-level identifier references appearing in expanded code are fully qualified identifiers exported by APIs, all top-level identifiers referred to in the definition of a syntax expander are visible at all use sites.

To ensure hygiene, all variables bound in a `SyntaxTree` resulting from an expansion are renamed, following the `syntax-case` system of Dybvig et al. [9].

36.4.2 Comments and Syntax Expansion

Comments are *not* recognized before syntax expansion (the embedded syntax may have its own commenting syntax): during expander scanning, an opening or terminating delimiter that occurs in what appears to be a Fortress comment is nonetheless recognized as an opening or terminating delimiter for that expander. Comments can be viewed as uses of a syntax expander, with special opening and terminating delimiters.

36.5 Expanders for Fortress

As the above examples demonstrate, it is often useful to denote Fortress abstract syntax using Fortress concrete syntax. A special set of syntax expanders are defined in the API `Fortress.Syntax` for every nonterminal in Fortress grammar, defined in Appendix G. The name of each expander consists of the name of the nonterminal in lowercase. The terminating symbol for each nonterminal is `end`. For example:

```
expr
  x + y
end
```

expands to the `SyntaxTree`:

```
Call(Empty,
     VarRef(Identifier("+")),
     VarRef(Identifier("x")),
     VarRef(Identifier("y")))
```

When one of these syntax expanders parses a binding construct, the bound identifier is replaced with an identifier resulting from a call to the function `gensym`, which yields an identifier distinct from all other identifiers bound in any component installed in the same fortress, or in any other fortress, anywhere, throughout all time past, present, and future. All variable references captured by the original identifier are replaced with references to the new identifier.

For convenience, a syntax expander with opening delimiter `<<` and terminating delimiter `>>` behaves identically to the `expr` expander when `Fortress.Syntax` is imported.

Part V

Fortress APIs and Documentation for Library Writers

Chapter 37

Algebraic Constraints

The traits in this component are used to describe properties of traits and their associated operators. These traits provide very few concrete methods, but specify abstract methods and property declarations.

37.1 Predicates and Equivalence Relations

A *predicate* is an operator that produces a boolean result. A binary predicate may be identified with a mathematical relation, where the predicate returns *true* in exactly those cases that its two operands satisfy the relation; therefore we use the mathematical terminology usually associated with relations to describe the properties of binary predicates.

```
trait UnaryPredicate[T extends UnaryPredicate[T, ~], opr ~]
  extends Any
  abstract opr ~(self): Boolean
end
```

A unary predicate is a prefix operator that takes one argument and returns a boolean value (*true* or *false*). Note that the symbol \sim is a static parameter, used here as a “variable” name for an operator.

```
trait BinaryPredicate[T extends BinaryPredicate[T, ~], opr ~]
  extends Any
  abstract opr ~(self, other: T): Boolean
end
```

A binary predicate is an infix operator that takes two arguments and returns a boolean value. Thus, for example, any trait T that extends `BinaryPredicate[T, \square]` necessarily has an infix method for the operator \square , and that operator returns a boolean value.

```
trait Reflexive[T extends Reflexive[T, ~], opr ~]
  extends { BinaryPredicate[T, ~] }
  property  $\forall(a: T) (a \sim a)$ 
end
```

A *reflexive* predicate always returns *true* when its operands are the same. Because this fact is expressed as a property declaration, the behavior of such an operator can be checked for correctness by unit testing.

```
trait Irreflexive[T extends Irreflexive[T, ~], opr ~]
```

```

    extends { BinaryPredicate[[T, ~]] }
  property  $\forall(a: T) \neg(a \sim a)$ 
end

```

An *irreflexive* predicate always returns *false* when its operands are the same. (Note that it is possible for a predicate to be neither reflexive nor irreflexive.)

```

  trait Symmetric[[T extends Symmetric[[T, ~]], opr ~]]
    extends { BinaryPredicate[[T, ~]] }
    property  $\forall(a: T, b: T) (a \sim b) \leftrightarrow (b \sim a)$ 
  end

```

A *symmetric* predicate doesn't care in which order its arguments are presented; the result is the same either way.

```

  trait Transitive[[T extends Transitive[[T, ~]], opr ~]]
    extends { BinaryPredicate[[T, ~]] }
    property  $\forall(a: T, b: T, c: T) ((a \sim b) \wedge (b \sim c)) \rightarrow (a \sim c)$ 
  end

```

A *transitive* predicate has the property that if *a* is related to *b* and *b* is related to *c*, then *a* is related to *c*.

```

  trait EquivalenceRelation[[T extends EquivalenceRelation[[T, ~]], opr ~]]
    extends { Reflexive[[T, ~]], Symmetric[[T, ~]], Transitive[[T, ~]] }
  end

```

An *equivalence relation* is any predicate that is reflexive, symmetric, and transitive. You can think of an equivalence relation as describing a way to separate a set of items into categories, such that each item belongs to exactly one category; the predicate is *true* of two items if and only if they are in the same category.

```

  trait IdentityEquality[[T extends IdentityEquality[[T]]]]
    extends { EquivalenceRelation[[T, =]] }
    opr =(self, other: T): Boolean
    property  $\forall(a: T, b: T) (\text{self} = \text{other}) \leftrightarrow (\text{self} \equiv \text{other})$ 
  end

```

This trait provides a concrete implementation of the operator `=` (defining it to behave the same as the operator `≡`) and states that `=` is an equivalence relation over instances of the type *T*.

```

  trait UnaryPredicateSubstitutionLaws[[T extends UnaryPredicateSubstitutionLaws[[T, ~, ≃]],
    opr ~, opr ≃]]
    extends { UnaryPredicate[[T, ~]], BinaryPredicate[[T, ≃]] }
    property  $\forall(a: T, a': T) (a \simeq a') \rightarrow ((\sim a) \leftrightarrow (\sim a'))$ 
  end

```

This handy trait states that the unary predicate `~` is consistent under substitutions described by the relation `≃` (which is typically, but not always, an equivalence relation); that is, the result produced by `~` is unchanged if its argument is replaced by some other value that is equivalent.

```

  trait BinaryPredicateSubstitutionLaws[[T extends BinaryPredicateSubstitutionLaws[[T, ~, ≃]],
    opr ~, opr ≃]]
    extends { BinaryPredicate[[T, ~]], BinaryPredicate[[T, ≃]] }
    property  $\forall(a: T, a': T) (a \simeq a') \rightarrow \forall(b: T) (a \sim b) \leftrightarrow (a' \sim b)$ 
    property  $\forall(b: T, b': T) (b \simeq b') \rightarrow \forall(a: T) (a \sim b) \leftrightarrow (a \sim b')$ 
  end

```

This equally handy trait states that the binary predicate \sim is consistent under substitutions described by the relation \simeq (which is typically, but not always, an equivalence relation); that is, the result produced by \sim is unchanged if either argument is replaced by some other value that is equivalent. (It is then easy to prove that the result is unchanged even when *both* arguments are replaced by equivalent values.)

37.2 Partial and Total Orders

```
trait Antisymmetric[T extends AntiSymmetric[T, ~], opr ~]
  extends { BinaryPredicate[T, ~], EquivalenceRelation[T, =],
           BinaryPredicateSubstitutionLaws[T, ~, =] }
  property  $\forall(a: T, b: T) ((a \sim b) \wedge (b \sim a)) \implies (a = b)$ 
end
```

A binary predicate \sim is *antisymmetric* if and only if two conditions are true: (a) \sim is consistent under substitutions described by the predicate $=$, which must be an equivalence relation; (b) whenever \sim holds true for a pair of arguments and for those same arguments in reverse order, those arguments are equivalent as specified by $=$.

```
trait PartialOrder[T extends PartialOrder[T, <], opr <]
  extends { Reflexive[T, <], Antisymmetric[T, <], Transitive[T, <] }
end
```

A *partial order* is a binary predicate that is reflexive, antisymmetric, and transitive.

```
trait StrictPartialOrder[T extends StrictPartialOrder[T, <], opr <]
  extends { Irreflexive[T, <], Antisymmetric[T, <], Transitive[T, <] }
end
```

A *strict partial order* is a binary predicate that is irreflexive, antisymmetric, and transitive. (Thus it differs from an ordinary partial order in being irreflexive rather than reflexive.)

It is easy to prove that, because $<$ is irreflexive and antisymmetric, that $a < b$ and $a = b$ cannot both be true. (If they were both true, then because antisymmetry requires that $<$ obey substitution laws, $a < a$ would be true—but that contradicts the fact that $<$ is irreflexive.) It is then easy to prove that $a < b$ and $b < a$ cannot both be true. (If they were both true, then by antisymmetry $a = b$ must be true—but $a < b$ and $a = b$ cannot both be true.)

```
trait TotalOrder[T extends TotalOrder[T, <], opr <]
  extends { PartialOrder[T, <] }
  property  $\forall(a: T, b: T) (a < b) \vee (b < a)$ 
end
```

A *total order* is a partial order in which every pair of operands must be related by the predicate $<$ in one order or the other; thus no two values are ever unordered with respect to each other.

```
trait StrictTotalOrder[T extends StrictTotalOrder[T, <], opr <]
  extends { StrictPartialOrder[T, <] }
  property  $\forall(a: T, b: T) (a < b) \vee (b < a) \vee (a = b)$ 
end
```

A *strict total order* is a strict partial order in which every pair of operands must be related, either by equality $=$ or by the predicate $<$ in one order or the other; thus no two values are ever unordered with respect to each other.

For a strict total order *at least* one of $a < b$ and $a = b$ and $b < a$ is true; but a strict total order is also a strict partial order, for which *at most* one of $a < b$ and $a = b$ and $b < a$ is true. Therefore a strict total order obeys the *law of trichotomy*: for any a and b , *exactly* one of $a < b$ and $a = b$ and $b < a$ is true.


```

trait PartialOrderOperators[[T extends PartialOrderOperators[[T, <, ≦, ≧, >, CMP]],
    opr <, opr ≦, opr ≧, opr >, opr CMP]]
  extends { StrictPartialOrder[[T, <]], PartialOrder[[T, ≦]],
    PartialOrder[[T, ≧]], StrictPartialOrder[[T, >]] }
  abstract opr CMP(self, other: T): Comparison
  property ∀(a: T, b: T) ((a CMP b) ≡ LessThan) ↔ ((b CMP a) ≡ GreaterThan)
  property ∀(a: T, b: T) ((a CMP b) ≡ EqualTo) ↔ ((b CMP a) ≡ EqualTo)
  property ∀(a: T, b: T) ((a CMP b) ≡ Unordered) ↔ ((b CMP a) ≡ Unordered)
  property ∀(a: T, b: T) (a < b) ↔ ((a CMP b) ≡ LessThan)
  property ∀(a: T, b: T) (a ≦ b) ↔ ((a CMP b) ≡ LessThan ∨ (a CMP b) ≡ EqualTo)
  property ∀(a: T, b: T) (a = b) ↔ ((a CMP b) ≡ EqualTo)
  property ∀(a: T, b: T) (a ≧ b) ↔ ((a CMP b) ≡ GreaterThan ∨ (a CMP b) ≡ EqualTo)
  property ∀(a: T, b: T) (a > b) ↔ ((a CMP b) ≡ GreaterThan)
end

```

For practical programming, we assume that partial order operators come in groups of five: a “less than” predicate, a “less than or equal to” predicate, a “greater than or equal to” predicate, a “greater than” predicate, and a “comparison” operator that returns one of the four values `LessThan`, `EqualTo`, `GreaterThan`, and `Unordered`. The trait `PartialOrderOperators` declares such a set of five operators and describes the necessary algebraic constraints among them and the equality predicate `=`.

```

trait TotalOrderOperators[[T extends TotalOrderOperators[[T, <, ≦, ≧, >, CMP]],
    opr <, opr ≦, opr ≧, opr >, opr CMP]]
  extends { PartialOrderOperators[[T, <, ≦, ≧, >, CMP]],
    StrictTotalOrder[[T, <]], TotalOrder[[T, ≦]],
    TotalOrder[[T, ≧]], StrictTotalOrder[[T, >]] }
  abstract opr CMP(self, other: T): TotalComparison
end

```

Total order operators likewise come in groups of five: a “less than” predicate, a “less than or equal to” predicate, a “greater than or equal to” predicate, a “greater than” predicate, and a “comparison” operator that returns one of the three values `LessThan`, `EqualTo`, and `GreaterThan`. The trait `TotalOrderOperators` declares such a set of five operators and describes the necessary algebraic constraints among them and the equality predicate `=`. For a total order, the comparison operator `CMP` never returns `Unordered`.

```

trait PartialOrderBasedOnLE[[T extends PartialOrderBasedOnLE[[T, <, ≦, ≧, >, CMP]],
    opr <, opr ≦, opr ≧, opr >, opr CMP]]
  extends { PartialOrderOperators[[T, <, ≦, ≧, >, CMP]] }
  opr CMP(self, other: T): Comparison
  opr <(self, other: T): Boolean
  opr =(self, other: T): Boolean
  opr ≧(self, other: T): Boolean
  opr >(self, other: T): Boolean
end

```

The trait `PartialOrderBasedOnLE` implements a partial order by assuming that the “less than or equal to” predicate is already defined and providing definitions for the comparison operator, the “less than” predicate, the equality predicate, the “greater than or equal to” predicate, and the “greater than” predicate in terms of the “less than or equal to” predicate.

```

trait TotalOrderBasedOnLE[[T extends TotalOrderBasedOnLE[[T, <, ≦, SUCCEQq, >, CMP]],
    opr <, opr ≦, opr ≧, opr >, opr CMP]]
  extends { TotalOrderOperators[[T, <, ≦, ≧, >, CMP]] }
  opr CMP(self, other: T): TotalComparison
  opr <(self, other: T): Boolean

```

```

opr =(self, other: T): Boolean
opr >(self, other: T): Boolean
opr >(self, other: T): Boolean
end

```

The trait `TotalOrderBasedOnLE` implements a total order by assuming that the “less than or equal to” predicate is already defined and providing definitions for the comparison operator, the “less than” predicate, the equality predicate, the “greater than or equal to” predicate, and the “greater than” predicate in terms of the “less than or equal to” predicate.

```

trait TotalOrderBasedOnLT[T extends TotalOrderBasedOnLT[T, <, <=, >, >=, CMP],
    opr <, opr <=, opr >, opr >=, opr CMP]
  extends { TotalOrderOperators[T, <, <=, >, >=, CMP] }
  opr CMP(self, other: T): TotalComparison
  opr <= (self, other: T): Boolean
  opr = (self, other: T): Boolean
  opr > (self, other: T): Boolean
  opr >= (self, other: T): Boolean
end

```

The trait `TotalOrderBasedOnLT` implements a total order by assuming that the “less than” predicate is already defined and providing definitions for the comparison operator, the “less than or equal to” predicate, the equality predicate, the “greater than or equal to” predicate, and the “greater than” predicate in terms of the “less than” predicate.

```

value object MaximalElement[opr <=] end
trait HasMaximalElement[T extends HasMaximalElement[T, <=], opr <=]
  extends { PartialOrder[T, <=] }
  where { T coerces MaximalElement[<=] }
  property ∀(a: T) a <= MaximalElement[<=]
end

```

The `HasMaximalElement` trait specifies that a partial order has a maximal element, that is, one to which every other element is related by the ordering predicate \preceq . The maximal element may be identified by coercing the object named `MaximalElement[<=]` to type T .

```

value object MinimalElement[opr <] end
trait HasMinimalElement[T extends HasMinimalElement[T, <], opr <]
  extends { PartialOrder[T, <] }
  where { T coerces MinimalElement[<] }
  property ∀(a: T) MinimalElement[<] < a
end

```

The `HasMinimalElement` trait specifies that a partial order has a minimal element, that is, one to which every other element relates by the ordering predicate \preceq . The minimal element may be identified by coercing the object named `MinimalElement[<]` to type T .

```

trait LexicographicPartialOrder[T extends LexicographicPartialOrder[T, ⊔, ⊑, ≡, ⊒, ⊓, TCMP,
    X, <, <=, ≃, >, >=, XCMP],
    opr ⊔, opr ⊑, opr ≡, opr ⊒, opr ⊓, opr TCMP,
    X extends TotalOrderOperators[X, <, <=, ≃, >, >=, XCMP],
    opr <, opr <=, opr ≃, opr >, opr >=, opr XCMP]
  extends { PartialOrderOperators[LexicographicPartialOrder[T, ⊔, ⊑, ≡, ⊒, ⊓, TCMP,
    X, <, <=, ≃, >, >=, XCMP],
    ⊔, ⊑, ⊒, ⊓, TCMP] }

```

```

    where { T extends ZeroBasedIndexing[[T, X]] }
    property  $\forall(a: T, b: T) (a \text{ TCMP } b) = (\text{BIG LEXICO}_{i \leftarrow a.\text{indices} \cap b.\text{indices}} (a_i \text{ XCMP } b_i)) \text{ LEXICO } (|a| \text{ CMP } |b|)$ 
end

```

The Comparison trait provides an associative operator LEXICO whose principal use is in defining lexicographic order on sequences of elements, which may be partial or total depending on whether the ordering of the elements is partial or total.

A set of lexicographic partial order operators $\sqsubset, \sqsubseteq, \equiv, \sqsupset, \sqsupseteq$, TCMP may be defined in terms of a partial order on the elements of the sequence with operators $\prec, \preceq, \simeq, \succ, \succeq$, XCMP. All that is really necessary is to define the lexicographic sequence comparison operator TCMP in terms of the element comparison operator XCMP; this is easily expressed by using the associative LEXICO operator to reduce the results of elementwise comparisons to a single value. (If the sequences to be compared are of unequal length, then the shorter sequence is compared to a prefix of the longer sequence, and if they are equal, then the longer sequence is considered to be greater than the shorter sequence. This rule is implemented by an additional application of the LEXICO operator to the result of comparing the lengths of the sequences.)

```

trait LexicographicTotalOrder[[T extends LexicographicTotalOrder[[T,  $\sqsubset, \sqsubseteq, \equiv, \sqsupset, \sqsupseteq$ , TCMP,
                                                                    X,  $\prec, \preceq, \simeq, \succ, \succeq$ , XCMP]],
                                opr  $\sqsubset, \text{opr } \sqsubseteq, \text{opr } \equiv, \text{opr } \sqsupset, \text{opr } \sqsupseteq, \text{opr } \text{TCMP},$ 
                                X extends TotalOrderOperators[[X,  $\prec, \preceq, \simeq, \succ, \succeq$ , XCMP]],
                                opr  $\prec, \text{opr } \preceq, \text{opr } \simeq, \text{opr } \succ, \text{opr } \succeq, \text{opr } \text{XCMP}]]$ 
  extends { LexicographicPartialOrder[[T,  $\sqsubset, \sqsubseteq, \equiv, \sqsupset, \sqsupseteq$ , TCMP, X,  $\prec, \preceq, \simeq, \succ, \succeq$ , XCMP]],
            TotalOrderOperators[[LexicographicTotalOrder[[T,  $\sqsubset, \sqsubseteq, \equiv, \sqsupset, \sqsupseteq$ , TCMP,
                                                            X,  $\prec, \preceq, \simeq, \succ, \succeq$ , XCMP]],
                                 $\sqsubset, \sqsubseteq, \sqsupset, \sqsupseteq, \text{TCMP}]]$  }
end

```

Similarly, a set of lexicographic total order operators $\sqsubset, \sqsubseteq, \equiv, \sqsupset, \sqsupseteq$, TCMP may be defined in terms of a total order on the elements of the sequence with operators $\prec, \preceq, \simeq, \succ, \succeq$, XCMP.

37.3 Operators and Their Properties

For some types, such as the integers \mathbb{Z} or the rationals \mathbb{Q} , results are always exact, and algebraic properties can be expected to be obeyed exactly. For other types, such as floating-point numbers, results are not always numerically exact, and algebraic properties can be expected to be obeyed only approximately. For example, given three floating-point values a and b and c , it may well be that $a + (b + c)$ is not equal to $(a + b) + c$; but we would expect their values to be reasonably close—unless, of course, overflow occurred in one expression but not the other.

In order to address the difficulties of such approximate computation, many of the traits described in this section come in two varieties: approximate and exact. The $+$ operator on integers or rationals is correctly described by the trait Associative, and the $+$ operator on floating-point numbers is correctly described by the trait ApproximatelyAssociative. An important distinction is that the predicate used to test acceptability of exact algebraic properties is $=$, which is required to be an equivalence relation and therefore transitive, but a type-dependent binary predicate (typically \approx) may be used to test acceptability of approximate algebraic properties, and this predicate is required only to be reflexive and symmetric.

```

trait UnaryOperator[[T extends UnaryOperator[[T,  $\odot$ ]], opr  $\odot$ ]
  extends Any
  abstract opr  $\odot(\text{self}): T$ 
end

```

A *unary operator* is a prefix operator that takes one argument and returns a value of the same type. Note that \odot is a static parameter, used here as a “variable” name for an operator.

```
trait BinaryOperator[T extends BinaryOperator[T, ⊙], opr ⊙]
  extends Any
  abstract opr ⊙(self, other: T): T
end
```

A *binary operator* is an infix operator that takes two arguments of the same type and returns a value of that type. Thus, for example, any trait T that extends `BinaryPredicate[T, +]` necessarily has an infix method for the operator $+$, and that operator takes two operands of type T and returns a value of type T .

```
trait IdentityOperator[T extends IdentityOperator[T]]
  extends { UnaryOperator[T, IDENTITY] }
  property ∀(a: T) (IDENTITY a) ≡ a
end
```

The trait `Object` extends `IdentityOperator[Object]`, so the `IDENTITY` operator is defined for every type whatsoever. (This operator may not be terribly useful for applications programming, but it has technical uses for specifying contracts and algebraic properties in libraries. It is used, for example, when defining the trait `BooleanAlgebra` in terms of the trait `Ring`: because every value is its own inverse with respect to the “exclusive OR” operator in a Boolean Algebra, `IDENTITY` is the appropriate additive inverse operator for use with the trait `Ring` in this connection.)

```
trait ApproximatelyCommutative[T extends ApproximatelyCommutative[T, ⊙, ≈], opr ⊙, opr ≈]
  extends { BinaryOperator[T, ⊙], Reflexive[T, ≈], Symmetric[T, ≈] }
  property ∀(a: T, b: T) (a ⊙ b) ≈ (b ⊙ a)
end
```

The trait `ApproximatelyCommutative` requires the operator \odot to be *approximately commutative*; that is, reversing the operands produces a result that is considered to be “close enough” as determined by the specified \approx predicate.

```
trait Commutative[T extends Commutative[T, ⊙], opr ⊙]
  extends { ApproximatelyCommutative[T, ⊙, =], EquivalenceRelation[T, =] }
end
```

The trait `Commutative` requires the operator \odot to be *commutative*; that is, reversing the operands produces an equal result.

```
trait ApproximatelyAssociative[T extends ApproximatelyAssociative[T, ⊙, ≈], opr ⊙, opr ≈]
  extends { BinaryOperator[T, ⊙], Reflexive[T, ≈], Symmetric[T, ≈] }
  property ∀(a: T, b: T, c: T) ((a ⊙ b) ⊙ c) ≈ (a ⊙ (b ⊙ c))
end
```

The trait `ApproximatelyAssociative` requires the operator \odot to be *approximately associative*; that is, the expressions $(a \odot b) \odot c$ and $a \odot (b \odot c)$ always produce results that are “close enough” to each other as determined by the specified \approx predicate.

```
trait Associative[T extends Associative[T, ⊙], opr ⊙]
  extends { ApproximatelyAssociative[T, ⊙, =], EquivalenceRelation[T, =] }
end
```

The trait `Associative` requires the operator \odot to be *associative*; that is, the expressions $(a \odot b) \odot c$ and $a \odot (b \odot c)$ always produce equal results.

```

trait IdempotentBinaryOperator[T extends IdempotentBinaryOperator[T, ⊙], opr ⊙]
  extends { BinaryOperator[T, ⊙], EquivalenceRelation[T, =] }
  property ∀(a: T) (a ⊙ a) :=: a
end

```

An idempotent binary operator has the property that if its two arguments are the same then the result is equal to each argument. For example, MAX and MIN are idempotent, as are \wedge and \vee applied to boolean arguments and \cap and \cup applied to sets; but $+$ applied to integers is not idempotent because $1 + 1$ does not produce 1 , and \oplus applied to boolean arguments is not idempotent because $true \oplus true$ produces $false$. The property of idempotency is sometimes of interest when performing reductions such as $\text{MAX}_{i \leftarrow 1:n} a_i$.

```

trait HasLeftIdentity[T extends HasLeftIdentity[T, ⊙], opr ⊙]
  extends { BinaryOperator[T, ⊙], EquivalenceRelation[T, =] }
  abstract isLeftIdentity(): Boolean
  property ∀(a: T, b: T) a.isLeftIdentity() →: ((a ⊙ b) = b)
end

```

A value e is a *left identity* for a binary operator \odot if the result of \odot always equals the right-hand operand whenever e is the left-hand operand. For example, 0 is a left identity for the $+$ operator on integers and the empty set is a left identity for the \cup operator on sets.

```

trait HasRightIdentity[T extends HasRightIdentity[T, ⊙], opr ⊙]
  extends { BinaryOperator[T, ⊙], EquivalenceRelation[T, =] }
  abstract isRightIdentity(): Boolean
  property ∀(a: T, b: T) b.isRightIdentity() →: ((a ⊙ b) = a)
end

```

A value e is a *right identity* for a binary operator \odot if the result of \odot always equals the right-hand operand whenever e is the left-hand operand. For example, 0 is a right identity (as well as a left identity) for the $+$ operator on integers and the empty set is a right identity (as well as a left identity) for the \cup operator on sets. By way of contrast, 1 is a right identity for division of rationals but not a left identity.

```

value object Identity[opr ⊙] end
trait HasIdentity[T extends HasIdentity[T, ⊙], opr ⊙]
  extends { HasLeftIdentity[T, ⊙], HasRightIdentity[T, ⊙] }
  where { T coerces Identity[⊙] }
  property ∀(a: T) (a ⊙ Identity[⊙]) = a
  property ∀(a: T) (Identity[⊙] ⊙ a) = a
  property ∀(a: T) a.isLeftIdentity() ↔ (a = Identity[⊙])
  property ∀(a: T) a.isRightIdentity() ↔ (a = Identity[⊙])
end

```

If the same value is both a left identity and a right identity for \odot , then it may be called simply an *identity*—in fact, *the identity*, for it is unique and may be obtained by coercing the object named `Identity[⊙]` to type T .

```

trait ApproximatelyHasInverses[T extends ApproximatelyHasInverses[T, ⊙, ⊘, ≈],
  opr ⊙, opr ⊘, opr ≈]
  extends { HasIdentity[T, ⊙], UnaryOperator[T, ⊘], BinaryOperator[T, ⊘] }
  where { T coerces Identity[⊙] }
  property ∀(a: T) (a ⊙ (⊘a)) :=: Identity[⊙]
  property ∀(a: T) ((⊘a) ⊙ a) :=: Identity[⊙]
  property ∀(a: T, b: T) (a ⊘ b) :=: (a ⊙ (⊘b))
end

```

A set of values with a binary operator \odot has *approximate inverses* if and only if the operator has an identity and for every value a there is another value a' such that the result of applying \odot to a and a' (in either order) is “close enough” to the identity. The unary operator \oslash returns the approximate inverse of its argument; as a notational convenience, it may also be used as a binary operator.

```
trait HasInverses[T extends HasInverses[[T, ⊙, ⊙], opr ⊙, opr ⊙]]
  extends { ApproximatelyHasInverses[[T, ⊙, ⊙, =]] }
  where { T coerces Identity[[⊙]] }
end
```

A set of values with a binary operator \odot has *inverses* if and only if the operator has an identity and for every value a there is another value a' such that the result of applying \odot to a and a' (in either order) equals the identity. The unary operator \oslash returns the inverse of its argument; as a notational convenience, it may also be used as a binary operator. A standard example is the operator $+$ on integers; the identity is 0 , and the unary operator $-$ returns the additive inverse of its argument, such that $a + (-a) = 0$ and $(-a) + a = 0$. Moreover, $-$ may be used as a binary operator: $a - b$ means $a + (-b)$.

```
trait HasLeftZeroes[T extends HasLeftZeroes[[T, ⊙, isLeftZero], opr ⊙, ident isLeftZero]]
  extends { BinaryOperator[[T, ⊙]] }
  abstract isLeftZero(): Boolean
  property ∀(a: T, b: T) a.isLeftZero() →: ((a ⊙ b) = a)
end
```

A value e is a *left zero* for a binary operator \odot if the result of \odot always equals e whenever e is the left-hand operand. For example, $-\infty$ is a left zero for the MIN operator on floating-point values, and $7FFFFFFF_{16}$ is a left zero for the MAX operator on values of type \mathbb{Z}_{32} . The purpose of this trait is to specify a method that says whether a given element is a left zero for \odot .

```
trait HasRightZeroes[T extends HasRightZeroes[[T, ⊙, isRightZero], opr ⊙, ident isRightZero]]
  extends { BinaryOperator[[T, ⊙]] }
  abstract isRightZero(): Boolean
  property ∀(a: T, b: T) b.isRightZero() →: ((a ⊙ b) = b)
end
```

A value e is a *right zero* for a binary operator \odot if the result of \odot always equals e whenever e is the right-hand operand. For example, $-\infty$ is a right zero (as well as a left zero) for the MIN operator on floating-point values, and $7FFFFFFF_{16}$ is a right zero (as well as a left zero) for the MAX operator on values of type \mathbb{Z}_{32} . By way of contrast, 0 is a left zero for the arithmetic shift operator on integers, but is not a right zero. The purpose of this trait is to specify a method that says whether a given element is a right zero for \odot .

```
trait ApproximatelyLeftDistributive[T extends ApproximatelyLeftDistributive[[T, ⊗, ⊕, ≈],
  opr ⊗, opr ⊕, opr ≈]]
  extends { BinaryOperator[[T, ⊗]], BinaryOperator[[T, ⊕]], Reflexive[[T, ≈]], Symmetric[[T, ≈]] }
  property ∀(a: T, b: T, c: T) (a ⊗ (b ⊕ c)) ≈: ((a ⊗ b) ⊕ (a ⊗ c))
end
```

The trait `ApproximatelyLeftDistributive` requires the operator \otimes to be *approximately left distributive* over the operator \oplus ; that is, the expressions $a \otimes (b \oplus c)$ and $(a \otimes b) \oplus (a \otimes c)$ always produce results that are “close enough” to each other as determined by the specified \approx predicate.

```
trait LeftDistributive[T extends LeftDistributive[[T, ⊗, ⊕], opr ⊗, opr ⊕]]
  extends { ApproximatelyLeftDistributive[[T, ⊗, ⊕, =]], EquivalenceRelation[[T, =]] }
end
```

The trait `LeftDistributive` requires the operator \otimes to be *left distributive* over the operator \oplus ; that is, the expressions $a \otimes (b \oplus c)$ and $(a \otimes b) \oplus (a \otimes c)$ always produce equal results.

```
trait ApproximatelyRightDistributive[T extends ApproximatelyRightDistributive[T, ⊗, ⊕, ≈],
  opr ⊗, opr ⊕, opr ≈]
  extends { BinaryOperator[T, ⊗], BinaryOperator[T, ⊕], Reflexive[T, ≈], Symmetric[T, ≈] }
  property ∀(a: T, b: T, c: T) ((a ⊕ b) ⊗ c) ≈: ((a ⊗ c) ⊕ (b ⊗ c))
end
```

The trait `ApproximatelyRightDistributive` requires the operator \otimes to be *approximately right distributive* over the operator \oplus ; that is, the expressions $(a \oplus b) \otimes c$ and $(a \otimes c) \oplus (b \otimes c)$ always produce results that are “close enough” to each other as determined by the specified \approx predicate.

```
trait RightDistributive[T extends RightDistributive[T, ⊗, ⊕], opr ⊗, opr ⊕]
  extends { ApproximatelyRightDistributive[T, ⊗, ⊕, =], EquivalenceRelation[T, =] }
end
```

The trait `RightDistributive` requires the operator \otimes to be *right distributive* over the operator \oplus ; that is, the expressions $(a \oplus b) \otimes c$ and $(a \otimes c) \oplus (b \otimes c)$ always produce equal results.

```
trait ApproximatelyDistributive[T extends ApproximatelyDistributive[T, ⊗, ⊕, ≈],
  opr ⊗, opr ⊕, opr ≈]
  extends { ApproximatelyLeftDistributive[T, ⊗, ⊕, ≈],
    ApproximatelyRightDistributive[T, ⊗, ⊕, ≈] }
end
```

The trait `ApproximatelyDistributive` requires the operator \otimes to be both approximately left distributive and approximately right distributive over the operator \oplus .

```
trait Distributive[T extends Distributive[T, ⊗, ⊕], opr ⊗, opr ⊕]
  extends { ApproximatelyDistributive[T, ⊗, ⊕, =], LeftDistributive[T, ⊗, ⊕], RightDistributive[T, ⊗, ⊕] }
end
```

The trait `Distributive` requires the operator \otimes to be both left distributive and right distributive over the operator \oplus .

```
trait AbsorptionLaws[T extends AbsorptionLaws[T, ∩, ∪], opr ∩, opr ∪]
  extends { BinaryOperator[T, ∩], BinaryOperator[T, ∪], EquivalenceRelation[T, =] }
  property ∀(a: T, b: T) a ∪ (a ∩ b) = a = a ∩ (a ∪ b)
end
```

The *absorption laws* specify certain relationships between two operators \cap and \cup . These laws are one of the defining properties of a lattice.

```
value object Zero[opr ⊗] end
trait ApproximateZeroAnnihilation[T extends ApproximateZeroAnnihilation[T, ⊗, ≈],
  opr ⊗, opr ≈]
  extends { BinaryOperator[T, ⊗], Reflexive[T, ≈], Symmetric[T, ≈] }
  where { T coerces Zero[⊗] }
  property ∀(a: T) (Zero[⊗] ⊗ a) ≈: Zero[⊗]
  property ∀(a: T) (a ⊗ Zero[⊗]) ≈: Zero[⊗]
end
```

An operator \otimes obeys *approximate zero annihilation* if and only if there is an element (call it z) that when used as either operand of \otimes causes the result to be “close enough” to z as determined by the specified \approx predicate. This zero element may be obtained by coercing the object named `Zero[⊗]` to type T .

```

trait ZeroAnnihilation[T extends ZeroAnnihilation[[T, ⊗], opr ⊗]]
  extends { ApproximateZeroAnnihilation[[T, ⊗, =]], EquivalenceRelation[[T, =]],
           HasLeftZeroes[[T, ⊗, isZero]], HasRightZeroes[[T, ⊗, isZero]] }
  where { T coerces Zero[[⊗]] }
end

```

An operator \otimes obeys *zero annihilation* if and only if there is an element (call it z) that when used as either operand of \otimes causes the result to equal z . This element is both a left zero and a right zero for \otimes .

```

trait NoApproximateZeroDivisors[T extends NoApproximateZeroDivisors[[T, ⊗, ≈], opr ⊗, opr ≈]]
  extends { BinaryOperator[[T, ⊗]], Reflexive[[T, ≈]], Symmetric[[T, ≈]] }
  where { T coerces Zero[[⊗]] }
  property ∀(a: T, b: T) (a ⊗ b ≈ Zero[[⊗]]) → (a ≈ Zero[[⊗]] ∨ b ≈ Zero[[⊗]])
end

```

An operator \otimes has *no approximate zero divisors* if and only if, for every pair of elements a and b , if $a \otimes b$ is approximately zero then at least one of a and b is approximately zero. (Great care is needed in the definition of “approximately” for this to work out correctly in practice.)

```

trait NoZeroDivisors[T extends NoZeroDivisors[[T, ⊗], opr ⊗]]
  extends { NoApproximateZeroDivisors[[T, ⊗, =]], EquivalenceRelation[[T, =]] }
  where { T coerces Zero[[⊗]] }
  property ∀(a: T, b: T) (a ⊗ b = Zero[[⊗]]) → (a = Zero[[⊗]] ∨ b = Zero[[⊗]])
end

```

An operator \otimes has *no zero divisors* if and only if, for every pair of elements a and b , if $a \otimes b$ is zero then at least one of a and b is zero (put conversely, if a and b are both nonzero then $a \otimes b$ must be nonzero).

```

trait UnaryOperatorSubstitutionLaws[T extends UnaryOperatorSubstitutionLaws[[T, ⊙, =],
  opr ⊙, opr ≈]]
  extends { UnaryOperator[[T, ⊙]], BinaryPredicate[[T, ≈]] }
  property ∀(a: T, a': T) (a ≈ a') → (⊙a) ≈ (⊙a')
end

```

This peculiarly spiffy trait states that the unary operator \odot is consistent under substitutions described by the relation \simeq (which is typically, but not always, an equivalence relation); that is, the result produced by \odot is unchanged if its argument is replaced by some other value that is equivalent.

```

trait BinaryOperatorSubstitutionLaws[T extends BinaryOperatorSubstitutionLaws[[T, ⊙, =],
  opr ⊙, opr ≈]]
  extends { BinaryOperator[[T, ⊙]], BinaryPredicate[[T, ≈]] }
  property ∀(a: T, a': T) (a ≈ a') → ∀(b: T) (a ⊙ b) ≈ (a' ⊙ b)
  property ∀(b: T, b': T) (b ≈ b') → ∀(a: T) (a ⊙ b) ≈ (a ⊙ b')
end

```

This equally spiffy trait states that the binary operator \odot is consistent under substitutions described by the relation \simeq (which is typically, but not always, an equivalence relation); that is, the result produced by \odot is unchanged if either argument is replaced by some other value that is equivalent. (It is then easy to prove that the result is unchanged even when *both* arguments are replaced by equivalent values.)

37.4 Lattices

```

trait Semilattice[T extends Semilattice[[T, ⊔], opr ⊔]]

```



```

    extends { Commutative[[T,  $\sqcap$ ]], Associative[[T,  $\sqcap$ ]], IdempotentBinaryOperator[[T,  $\sqcap$ ]] }
end

```

A *semilattice* is a set of values with an operator \sqcap that is commutative, associative, and idempotent. Examples of such operators are \wedge , \vee , \cap , \cup , MAX, MIN, GCD, and LCM.

```

trait BoundedSemilattice[[T extends BoundedSemilattice[[T,  $\sqcap$ ]], opr  $\sqcap$ ]
  extends { Semilattice[[T,  $\sqcap$ ]], CommutativeMonoid[[T,  $\sqcap$ ]] }
  where { T coerces Identity[[ $\sqcap$ ]] }
end

```

A *bounded semilattice* is a semilattice for which the operator \sqcap has an identity. For example, the boolean value *true* is the identity for \wedge and the empty set is the identity for \cup .

```

trait Lattice[[T extends Lattice[[T,  $\sqcap$ ,  $\sqcup$ ]], opr  $\sqcap$ , opr  $\sqcup$ ]
  extends { Semilattice[[T,  $\sqcap$ ]], Semilattice[[T,  $\sqcup$ ]], AbsorptionLaws[[T,  $\sqcap$ ,  $\sqcup$ ]] }
end

```

A *lattice* is a pair of semilattices that share the same set of values and whose operators together obey the absorption laws. One operator (here, by convention, \sqcap) is called the *meet* operator and the other (here, by convention, \sqcup) is called the *join* operator.

```

trait MeetBoundedLattice[[T extends MeetBoundedLattice[[T,  $\sqcap$ ,  $\sqcup$ ]], opr  $\sqcap$ , opr  $\sqcup$ ]
  extends { Lattice[[T,  $\sqcap$ ,  $\sqcup$ ]], BoundedSemilattice[[T,  $\sqcap$ ]] }
  where { T coerces Identity[[ $\sqcap$ ]] }
end

```

If the semilattice associated with the meet operator of a lattice is bounded, then the lattice is a *meet-bounded lattice*.

```

trait JoinBoundedLattice[[T extends JoinBoundedLattice[[T,  $\sqcap$ ,  $\sqcup$ ]], opr  $\sqcap$ , opr  $\sqcup$ ]
  extends { Lattice[[T,  $\sqcap$ ,  $\sqcup$ ]], BoundedSemilattice[[T,  $\sqcup$ ]] }
  where { T coerces Identity[[ $\sqcup$ ]] }
end

```

If the semilattice associated with the join operator of a lattice is bounded, then the lattice is a *join-bounded lattice*.

```

trait BoundedLattice[[T extends BoundedLattice[[T,  $\sqcap$ ,  $\sqcup$ ]], opr  $\sqcap$ , opr  $\sqcup$ ]
  extends { MeetBoundedLattice[[T,  $\sqcap$ ,  $\sqcup$ ]], JoinBoundedLattice[[T,  $\sqcap$ ,  $\sqcup$ ]] }
  where { T coerces Identity[[ $\sqcap$ ]], T coerces Identity[[ $\sqcup$ ]] }
end

```

If a lattice is both meet-bounded and join-bounded, then we simply say that the lattice is *bounded*.

```

trait PartialOrderAndLattice[[T extends PartialOrderLattice[[T,  $\preceq$ ,  $\sqcap$ ,  $\sqcup$ ]],
  opr  $\preceq$ , opr  $\sqcap$ , opr  $\sqcup$ ]
  extends { PartialOrder[[T,  $\preceq$ ]], Lattice[[T,  $\sqcap$ ,  $\sqcup$ ]] }
  property  $\forall(a: T, b: T) (a \preceq b) \leftrightarrow (a \sqcap b = a)$ 
  property  $\forall(a: T, b: T) (a \preceq b) \leftrightarrow (a \sqcup b = b)$ 
end

```

Every lattice defines a partial order, and every partial order defines a lattice. This trait specifies that the type T provides the necessary operators to be regarded as both a partial order and a lattice, states the properties that should relate the partial-order behavior to the lattice behavior. As an example, if the set of values is the integers \mathbb{Z} , the partial order operator is LEQ, and the lattice operators are MIN and MAX, then the requirements are that $a \text{ LEQ } b$ if and only if $a = a \text{ MIN } b$, and that $a \text{ LEQ } b$ if and only if $b = a \text{ MAX } b$.

```

trait PartialOrderAndMeetBoundedLattice[T extends PartialOrderAndMeetBoundedLattice[T, ≤, ⊔, ⊓],
    opr ≤, opr ⊔, opr ⊓]
  extends { PartialOrderAndLattice[T, ≤, ⊔, ⊓],
    MeetBoundedLattice[T, ⊔, ⊓],
    HasMinimalElement[T, ≤] }
  where { T coerces Identity[⊔], T coerces MaximalElement[≤] }
  property T.coercion (Identity[⊔]) = T.coercion (MaximalElement[≤])
end

```

The partial order associated with a meet-bounded lattice should have a maximal element, which is the identity of the meet operator.

```

trait PartialOrderAndJoinBoundedLattice[T extends PartialOrderAndJoinBoundedLattice[T, ≤, ⊔, ⊓],
    opr ≤, opr ⊔, opr ⊓]
  extends { PartialOrderAndLattice[T, ≤, ⊔, ⊓],
    JoinBoundedLattice[T, ⊔, ⊓],
    HasMaximalElement[T, ≤] }
  where { T coerces Identity[⊓], T coerces MinimalElement[≤] }
  property T.coercion (Identity[⊓]) = T.coercion (MinimalElement[≤])
end

```

The partial order associated with a join-bounded lattice should have a minimal element, which is the identity of the join operator.

```

trait PartialOrderAndBoundedLattice[T extends PartialOrderAndBoundedLattice[T, ≤, ⊔, ⊓],
    opr ≤, opr ⊔, opr ⊓]
  extends { PartialOrderAndMeetBoundedLattice[T, ≤, ⊔, ⊓],
    PartialOrderAndJoinBoundedLattice[T, ≤, ⊔, ⊓] }
end

```

The partial order associated with a bounded lattice should have both a minimal element and a maximal element.

37.5 Monoids, Groups, Rings, and Fields

```

trait ApproximateMonoid[T extends ApproximateMonoid[T, ⊙, ≈], opr ⊙, opr ≈]
  extends { ApproximatelyAssociative[T, ⊙, ≈], HasIdentity[T, ⊙] }
  where { T coerces Identity[⊙] }
end

```

An *approximate monoid* is a set of values with an approximately associative binary operator \odot that has an identity. For example, floating-point multiplication has identity 1 and is approximately associative.

```

trait Monoid[T extends Monoid[T, ⊙], opr ⊙]
  extends { ApproximateMonoid[T, ⊙, =], Associative[T, ⊙] }
  where { T coerces Identity[⊙] }
end

```

A *monoid* is a set of values with an associative binary operator \odot that has an identity. For example, trait `String` extends `Monoid[String, ||]` where `||` is the string concatenation operator. Note that string concatenation is associative but not commutative and that the empty string is the identity for string concatenation, so coercing `Identity[||]` to type `String` produces the empty string.

```

trait ApproximateCommutativeMonoid[T extends ApproximateCommutativeMonoid[[T, ⊕, ≈]],
                                opr ⊕, opr ≈]
  extends { ApproximateMonoid[[T, ⊕, ≈]], ApproximatelyCommutative[[T, ⊕, ≈]] }
  where { T coerces Identity[[⊕]] }
end

```

An *approximate commutative monoid* is an approximate monoid whose binary operator is also approximately commutative. For example, floating-point multiplication has identity 1 and is approximately associative and also approximately (indeed, exactly) commutative.

```

trait CommutativeMonoid[T extends CommutativeMonoid[[T, ⊕]], opr ⊕]
  extends { ApproximateCommutativeMonoid[[T, ⊕, =]], Monoid[[T, ⊕]], Commutative[[T, ⊕]] }
  where { T coerces Identity[[⊕]] }
end

```

A *commutative monoid* is a monoid whose binary operator is also commutative. For example, the operator \wedge on boolean values is associative and commutative and has identity *true*; likewise, the operator \vee on boolean values is associative and commutative and has identity *false*.

```

trait ApproximateGroup[T extends ApproximateGroup[[T, ⊙, ⊘, ≈]], opr ⊙, opr ⊘, opr ≈]
  extends { ApproximateMonoid[[T, ⊙, ≈]], ApproximatelyHasInverses[[T, ⊙, ⊘, ≈]] }
  where { T coerces Identity[[⊙]] }
end

```

An *approximate group* is an approximate monoid that has approximate inverses. For example, a floating-point representation of quaternions with multiplication as the binary operator would form an approximate group.

```

trait Group[T extends Group[[T, ⊙, ⊘]], opr ⊙, opr ⊘]
  extends { ApproximateGroup[[T, ⊙, ⊘, =]], Monoid[[T, ⊙]], HasInverses[[T, ⊙, ⊘]] }
  where { T coerces Identity[[⊙]] }
end

```

A *group* is monoid that has inverses.

```

trait ApproximateAbelianGroup[T extends ApproximateAbelianGroup[[T, ⊕, ⊖, ≈]],
                              opr ⊕, opr ⊖, opr ≈]
  extends { ApproximateGroup[[T, ⊕, ⊖, ≈]],
            ApproximateCommutativeMonoid[[T, ⊕, ≈]] }
  where { T coerces Identity[[⊕]] }
end

```

An *approximate Abelian group* is an approximate group whose binary operator is also approximately commutative.

```

trait AbelianGroup[T extends AbelianGroup[[T, ⊕, ⊖]], opr ⊕, opr ⊖]
  extends { ApproximateAbelianGroup[[T, ⊕, ⊖, =]],
            Group[[T, ⊕, ⊖]], CommutativeMonoid[[T, ⊕]] }
  where { T coerces Identity[[⊕]] }
end

```

An *Abelian group* is group whose binary operator is also commutative. (The term “Abelian” is traditionally used instead of “commutative” when discussing groups, in tribute to mathematician Niels Henrik Abel.) For example, the integers with the binary addition operator $+$, unary negation operator $-$, and identity 0 form an Abelian group. As another example, the boolean values with the binary exclusive OR operator \oplus and unary negation operator `IDENTITY` form an Abelian group; the value *false* is the identity for \oplus .

```

trait ApproximateSemiRing[T extends ApproximateSemiRing[T, ⊕, ⊗, ≈],
  opr ⊕, opr ⊗, opr ≈]
  extends { ApproximateCommutativeMonoid[T, ⊕, ≈],
    ApproximateMonoid[T, ⊗, ≈],
    ApproximatelyDistributive[T, ⊗, ⊕, ≈],
    ApproximateZeroAnnihilation[T, ⊗, ≈] }
  where { T coerces Identity[⊕], T coerces Identity[⊗], T coerces Zero[⊗] }
  property T.coercion (Zero[⊗]) ≈ T.coercion (Identity[⊕])
end

```

An *approximate semiring* is a set of values that has two binary operators \oplus and \otimes , such that (a) the values form an approximate commutative monoid with \oplus ; (b) the values form an approximate monoid with \otimes ; (c) \otimes is approximately distributive over \oplus ; and (d) \otimes obeys approximate zero annihilation, where the zero for \otimes is the same as the identity for \oplus .

```

trait SemiRing[T extends SemiRing[T, ⊕, ⊗], opr ⊕, opr ⊗]
  extends { ApproximateSemiRing[T, ⊕, ⊗, =],
    CommutativeMonoid[T, ⊕],
    Monoid[T, ⊗],
    Distributive[T, ⊗, ⊕],
    ZeroAnnihilation[T, ⊗] }
  where { T coerces Identity[⊕], T coerces Identity[⊗], T coerces Zero[⊗] }
end

```

A *semiring* is a set of values that has two binary operators \oplus and \otimes , such that (a) the values form a commutative monoid with \oplus ; (b) the values form a monoid with \otimes ; (c) \otimes is distributive over \oplus ; and (d) \otimes obeys zero annihilation, where the zero for \otimes is the same as the identity for \oplus .

```

trait ApproximateRing[T extends ApproximateRing[T, ⊕, ⊖, ⊗, ≈],
  opr ⊕, opr ⊖, opr ⊗, opr ≈]
  extends { ApproximateAbelianGroup[T, ⊕, ⊖, ≈],
    ApproximateSemiRing[T, ⊕, ⊗, ≈] }
  where { T coerces Identity[⊕], T coerces Identity[⊗], T coerces Zero[⊗] }
end

```

An *approximate ring* is an approximate semiring that also has a unary operator \ominus that returns inverses for the \oplus operator so that the values form an approximate group with \oplus and \ominus .

```

trait Ring[T extends Ring[T, ⊕, ⊖, ⊗], opr ⊕, opr ⊖, opr ⊗]
  extends { ApproximateRing[T, ⊕, ⊖, ⊗, =],
    AbelianGroup[T, ⊕, ⊖],
    SemiRing[T, ⊕, ⊗] }
  where { T coerces Identity[⊕], T coerces Identity[⊗], T coerces Zero[⊗] }
end

```

A *ring* is a semiring that also has a unary operator \ominus that returns inverses for the \oplus operator so that the values form a group with \oplus and \ominus .

```

trait ApproximateCommutativeRing[T extends ApproximateCommutativeRing[T, ⊕, ⊖, ⊗, ≈],
  opr ⊕, opr ⊖, opr ⊗, opr ≈]
  extends { ApproximateRing[T, ⊕, ⊖, ⊗, ≈],
    ApproximateCommutativeMonoid[T, ⊗, ≈] }
  where { T coerces Identity[⊕], T coerces Identity[⊗], T coerces Zero[⊗] }
end

```

An *approximate commutative ring* is an approximate ring for which the binary operator \otimes is also approximately commutative.

```

trait CommutativeRing[T extends CommutativeRing[T, ⊕, ⊖, ⊗],
  opr ⊕, opr ⊖, opr ⊗]
  extends { ApproximateCommutativeRing[T, ⊕, ⊖, ⊗, =],
    Ring[T, ⊕, ⊖, ⊗],
    CommutativeMonoid[T, ⊗] }
  where { T coerces Identity[⊕], T coerces Identity[⊗], T coerces Zero[⊗] }
end

```

A *commutative ring* is a ring for which the binary operator \otimes is also commutative.

```

trait ApproximateMathematicalDomain[T extends ApproximateRing[T, ⊕, ⊖, ⊗, ≈],
  opr ⊕, opr ⊖, opr ⊗, opr ≈]
  extends { ApproximateRing[T, ⊕, ⊖, ⊗, ≈],
    NoApproximateZeroDivisors[T, ⊗, ≈] }
  where { T coerces Identity[⊕], T coerces Identity[⊗], T coerces Zero[⊗] }
end

```

An *approximate mathematical domain* is an approximate ring that has no approximate zero divisors.

```

trait MathematicalDomain[T extends Ring[T, ⊕, ⊖, ⊗],
  opr ⊕, opr ⊖, opr ⊗]
  extends { ApproximateMathematicalDomain[T, ⊕, ⊖, ⊗, =],
    Ring[T, ⊕, ⊖, ⊗],
    NoZeroDivisors[T, ⊗] }
  where { T coerces Identity[⊕], T coerces Identity[⊗], T coerces Zero[⊗] }
end

```

A *mathematical domain* is a ring that has no zero divisors. (In mathematics, this is called simply a *domain*. In Fortress, the term “mathematical domain” is used instead to avoid consuming this word for a concept that admittedly is used rarely in programming when compared with other common single words such as “ring” and “field.”)

```

trait ApproximateIntegralDomain[T extends ApproximateRing[T, ⊕, ⊖, ⊗, ≈],
  opr ⊕, opr ⊖, opr ⊗, opr ≈]
  extends { ApproximateMathematicalDomain[T, ⊕, ⊖, ⊗, ≈],
    ApproximateCommutativeRing[T, ⊕, ⊖, ⊗, ≈] }
  where { T coerces Identity[⊕], T coerces Identity[⊗], T coerces Zero[⊗] }
end

```

An *approximate integral domain* is an approximate mathematical domain for which the binary operator \otimes is also approximately commutative.

```

trait IntegralDomain[T extends Ring[T, ⊕, ⊖, ⊗],
  opr ⊕, opr ⊖, opr ⊗]
  extends { ApproximateIntegralDomain[T, ⊕, ⊖, ⊗, =],
    MathematicalDomain[T, ⊕, ⊖, ⊗],
    CommutativeRing[T, ⊕, ⊖, ⊗] }
  where { T coerces Identity[⊕], T coerces Identity[⊗], T coerces Zero[⊗] }
end

```

An *integral domain* is a mathematical domain for which the binary operator \otimes is also commutative.

```

trait ApproximateDivisionRing[T extends ApproximateDivisionRing[T, U, ⊕, ⊖, ⊗, ⊘, ≈],
    U extends T,
    opr ⊕, opr ⊖, opr ⊗, opr ⊘, opr ≈]
  extends { ApproximateMathematicalDomain[T, ⊕, ⊖, ⊗, ≈],
    ApproximateGroup[U, ⊗, ⊘, ≈] }
  where { T coerces Identity[⊕], T coerces Identity[⊗], T coerces Zero[⊗] }
  property ¬instanceOf[U](T. coercion (Zero[⊕]))
  property ∀(a:T) a ≠ Zero[⊕] → instanceOf[U](a)
end

```

An *approximate division ring* is an approximate ring for which the binary operator \otimes also has approximate inverses, so that the values other than the zero of \otimes form an approximate group with \otimes and \oslash . An approximate division ring is in fact also an approximate mathematical domain.

```

trait DivisionRing[T extends DivisionRing[T, U, ⊕, ⊖, ⊗, ⊘],
    U extends T,
    opr ⊕, opr ⊖, opr ⊗, opr ⊘]
  extends { ApproximateDivisionRing[T, U, ⊕, ⊖, ⊗, ⊘, =],
    MathematicalDomain[T, ⊕, ⊖, ⊗, ≈],
    Group[U, ⊗, ⊘] }
  where { T coerces Identity[⊕], T coerces Identity[⊗], T coerces Zero[⊗] }
end

```

A *division ring* is a ring for which the binary operator \otimes also has inverses, so that the values other than the zero of \otimes form a group with \otimes and \oslash . A division ring is in fact also a mathematical domain.

```

trait ApproximateField[T extends ApproximateField[T, U, ⊕, ⊖, ⊗, ⊘, ≈],
    U extends T,
    opr ⊕, opr ⊖, opr ⊗, opr ⊘, opr ≈]
  extends { ApproximateIntegralDomain[T, ⊕, ⊖, ⊗, ≈],
    ApproximateDivisionRing[T, U, ⊕, ⊖, ⊗, ⊘, ≈],
    ApproximateAbelianGroup[T, ⊗, ⊘, ≈] }
  where { T coerces Identity[⊕], T coerces Identity[⊗], T coerces Zero[⊗] }
end

```

An *approximate field* is an approximately commutative ring that is also an approximate division ring: the binary operator \otimes is approximately commutative and also has approximate inverses, so that the values other than the zero of \otimes form an approximate Abelian group with \otimes and \oslash . An approximate field is in fact also an approximate integral domain.

```

trait Field[T extends Field[T, U, ⊕, ⊖, ⊗, ⊘], U extends T, opr ⊕, opr ⊖, opr ⊗, opr ⊘]
  extends { ApproximateField[T, U, ⊕, ⊖, ⊗, ⊘, =],
    IntegralDomain[T, ⊕, ⊖, ⊗],
    DivisionRing[T, U, ⊕, ⊖, ⊗, ⊘],
    AbelianGroup[T, ⊗, ⊘] }
  where { T coerces Identity[⊕], T coerces Identity[⊗], T coerces Zero[⊗] }
end

```

A *field* is a commutative ring that is also a division ring: the binary operator \otimes is commutative and also has inverses, so that the values other than the zero of \otimes form an Abelian group with \otimes and \oslash . A field is in fact also an integral domain.

37.6 Boolean Algebras

```

value object ComplementBound[opr ⊙] end
trait HasComplements[T extends HasComplements[[T, ⊙, ~], opr ⊙, opr ~]]
  extends { BinaryOperator[[T, ⊙], UnaryOperator[[T, ~], EquivalenceRelation[[T, =]] }
  where { T coerces ComplementBound[[⊙]] }
  property ∀(a: T) (a ⊙ (~ a)) := ComplementBound[[⊙]]
end

```

A set of values with a binary operator \odot has *complements* if and only if there is a specific value e such that for every value a there is another value a' such that the result of applying \odot to a and a' (in either order) equals the specified value e . This value may be obtained by coercing the object named `ComplementBound[[⊙]]` to type T . The unary operator \sim returns the complement of its argument with respect to the operator \odot .

Note that the trait `HasComplements` is similar to the trait `HasInverses`, but `HasComplements` does not require that that specified element be an identity of the binary operator.

```

trait DeMorgan[T extends DeMorgan[[T, ∧, ∨, ~], opr ∧, opr ∨, opr ~]]
  extends { BinaryOperator[[T, ∧], BinaryOperator[[T, ∨],
    UnaryOperator[[T, ~], EquivalenceRelation[[T, =]] }
  property ∀(a: T, b: T) (~ (a ∨ b)) := ((~ a) ∧ (~ b))
end

```

This trait expresses De Morgan's law for two binary operators \wedge and \vee and a unary operator \sim : the expressions $\sim (a \vee b)$ and $(\sim a) \wedge (\sim b)$ produce equal results.

```

trait BooleanAlgebra[T extends BooleanAlgebra[[T, ∧, ∨, ~, ⊔], opr ∧, opr ∨, opr ~, opr ⊔]]
  extends { Commutative[[T, ∧], Associative[[T, ∧],
    Commutative[[T, ∨], Associative[[T, ∨],
    IdempotentBinaryOperator[[T, ∧],
    IdempotentBinaryOperator[[T, ∨],
    HasIdentity[[T, ∧], HasIdentity[[T, ∨],
    HasComplements[[T, ∨, ~], HasComplements[[T, ∧, ~],
    Distributive[[T, ∧, ∨], Distributive[[T, ∨, ∧],
    DeMorgan[[T, ∧, ∨, ~], DeMorgan[[T, ∨, ∧, ~],
    BoundedLattice[[T, ∧, ∨], Ring[[T, ⊔, IDENTITY, ∧]] }
  where { T coerces Identity[[∧], T coerces ComplementBound[[∧],
    T coerces Identity[[∨], T coerces ComplementBound[[∨],
    T coerces Identity[[⊔], T coerces Zero[[∧]] }
  property ∀(a: T) (~ (~ a)) := a
  property ∀(a: T, b: T) a ⊔ b = (a ∧ (~ b)) ∨ ((~ a) ∧ b)
  property T.coercion (Identity[[⊔]]) = T.coercion (Identity[[∨]])
  property T.coercion (ComplementBound[[∧]]) = T.coercion (Identity[[∨]])
  property T.coercion (ComplementBound[[∨]]) = T.coercion (Identity[[∧]])
end

```

A *boolean algebra* is an algebraic structure consisting of a set of values, three binary operators \wedge , \vee , and \sqcup , and a unary operator \sim , such that the operators obey a number of specific properties:

- \wedge is commutative, associative, and idempotent, and has a unique identity
- \vee is commutative, associative, and idempotent, and has a unique identity
- \wedge has complements with respect to \sim
- \vee has complements with respect to \sim

- \wedge and \vee distribute over each other
- De Morgan's law applies to \wedge , \vee , and \sim , and also to \vee , \wedge , and \sim
- The values form a bounded lattice with \wedge as the meet operator and \vee as the join operator.
- The values form a ring with $\underline{\vee}$ as the addition operator, IDENTITY as the additive inverse operator, and \wedge as the multiplication operator

The type `Boolean` is the most familiar example of a boolean algebra. The power set of a set (that is, the set of all subsets of the set) also forms a boolean algebra with operators \cap , \cup , set complement, and symmetric set difference; the empty set is the identity for \cup , and the original set is the identity for \cap .

Chapter 38

Numbers

38.1 The Trait `Fortress.Standard.RationalQuantity`

The standard types for rational numbers such as \mathbb{Q} and \mathbb{Q}^* and $\mathbb{Q}_{\leq}^{\#}$ are defined in terms of a single trait `RationalQuantity` that handles dimensions and units as well as performing a case analysis to distinguish rather a particular expression is guaranteed not to produce infinities, $0/0$, or numbers of a particular sign.

The trait `RationalQuantity` takes seven static parameters; the first is a dimensional unit, and the others are booleans specifying whether an instance of the trait can possibly be $-\infty$, a finite rational less than zero, zero, a finite rational greater than zero, $+\infty$, or $0/0$. This allows the standard rational types to be represented as follows:

```
type Q = RationalQuantity[[dimensionless, false, true, true, true, false, false]]
type Q< = RationalQuantity[[dimensionless, false, true, false, false, false, false]]
type Q<= = RationalQuantity[[dimensionless, false, true, true, false, false, false]]
type Q>= = RationalQuantity[[dimensionless, false, false, true, true, false, false]]
type Q> = RationalQuantity[[dimensionless, false, false, false, true, false, false]]
type Q≠ = RationalQuantity[[dimensionless, false, true, false, true, false, false]]
type Q* = RationalQuantity[[dimensionless, true, true, true, true, true, false]]
type Q*< = RationalQuantity[[dimensionless, true, true, false, false, false, false]]
type Q*<= = RationalQuantity[[dimensionless, true, true, true, false, false, false]]
type Q*>= = RationalQuantity[[dimensionless, false, false, true, true, true, false]]
type Q*> = RationalQuantity[[dimensionless, false, false, false, true, true, false]]
type Q*≠ = RationalQuantity[[dimensionless, true, true, false, true, true, false]]
type Q# = RationalQuantity[[dimensionless, true, true, true, true, true, true]]
type Q#< = RationalQuantity[[dimensionless, true, true, false, false, false, true]]
type Q#<= = RationalQuantity[[dimensionless, true, true, true, false, false, true]]
type Q#>= = RationalQuantity[[dimensionless, false, false, true, true, true, true]]
type Q#> = RationalQuantity[[dimensionless, false, false, false, true, true, true]]
type Q#≠ = RationalQuantity[[dimensionless, true, true, false, true, true, true]]
```

Here is the detailed description of `RationalQuantity`, showing the details of the type calculations:

```

trait RationalQuantity[[unit U absorbs unit,
  bool ninf, bool lt, bool eq, bool gt, bool pinf, bool nan]]
  extends { RationalQuantity[[U, ninf', lt', eq', gt', pinf', nan']]
    where { bool ninf', bool lt', bool eq', bool gt', bool pinf', bool nan',
      ninf → ninf', lt → lt', eq → eq', gt → gt', pinf → pinf', nan → nan' },
    Field[[ RationalQuantity[[U, ninf, lt, eq, gt, pinf, nan]],
      RationalQuantity[[U, ninf, lt, false, gt, pinf, nan]], +, -, ·, /]]
    where { lt ∧ eq ∧ gt ∧ ¬ninf ∧ ¬pinf ∧ ¬nan, U = dimensionless },
    Field[[ RationalQuantity[[U, ninf, lt, eq, gt, pinf, nan]],
      RationalQuantity[[U, ninf, lt, false, gt, pinf, nan]], +, -, ×, /]]
    where { lt ∧ eq ∧ gt ∧ ¬ninf ∧ ¬pinf ∧ ¬nan, U = dimensionless },
    Field[[ RationalQuantity[[U, ninf, lt, eq, gt, pinf, nan]],
      RationalQuantity[[U, ninf, lt, false, gt, pinf, nan]], +, -, juxtaposition, /]]
    where { lt ∧ eq ∧ gt ∧ ¬ninf ∧ ¬pinf ∧ ¬nan, U = dimensionless },
    AbelianGroup[[RationalQuantity[[U, ninf, lt, eq, gt, pinf, nan]], +, -]],
    TotalOrderOperators[[RationalQuantity[[U, ninf, lt, eq, gt, pinf, nan]], <, ≤, ≥, >, CMP]]
    where { ¬nan },
    PartialOrderAndLattice[[RationalQuantity[[U, ninf, lt, eq, gt, pinf, nan]], ≤, MIN, MAX]]
    where { ¬nan },
    PartialOrderAndMeetBoundedLattice[[ RationalQuantity[[U, ninf, lt, eq, gt, pinf, nan]],
      ≤, MIN, MAX]]

    where { pinf ∧ ¬nan },
    PartialOrderAndJoinBoundedLattice[[ RationalQuantity[[U, ninf, lt, eq, gt, pinf, nan]],
      ≤, MIN, MAX]]

    where { ninf ∧ ¬nan },
    PartialOrderAndBoundedLattice[[ RationalQuantity[[U, ninf, lt, eq, gt, pinf, nan]],
      ≤, MIN, MAX]]

    where { ninf ∧ pinf ∧ ¬nan } }
  where { ninf ∨ lt ∨ eq ∨ gt ∨ pinf ∨ nan }
  coercion (x: Identity[[+]])
  coercion (x: Identity[[·]])
  coercion (x: Identity[[×]])
  coercion (x: Identity[[juxtaposition]])
  coercion (x: Zero[[·]])
  coercion (x: Zero[[×]])
  coercion (x: Zero[[juxtaposition]])
  coercion (x: IntegerQuantity[[U, ninf, lt, eq, gt, pinf, nan]])
  opr juxtaposition [[unit U', bool ninf', bool lt', bool eq', bool gt', bool pinf', bool nan']]
    (self, other: RationalQuantity[[U', ninf', lt', eq', gt', pinf', nan']]):
    RationalQuantity[[U U',
      ninf ∧ pinf' ∨ ninf ∧ gt' ∨ lt ∧ pinf' ∨ pinf ∧ ninf' ∨ pinf ∧ lt' ∨ gt ∧ ninf',
      lt ∧ gt' ∨ gt ∧ lt',
      eq ∧ (lt' ∨ eq' ∨ gt') ∨ (lt ∨ eq ∨ gt) ∧ eq',
      lt ∧ lt' ∨ gt ∧ gt',
      ninf ∧ ninf' ∨ ninf ∧ lt' ∨ lt ∧ ninf' ∨ pinf ∧ pinf' ∨ pinf ∧ gt' ∨ gt ∧ pinf',
      nan ∨ nan' ∨ ninf ∧ eq' ∨ pinf ∧ eq' ∨ eq ∧ ninf' ∨ eq ∧ pinf']]
  opr +(self): RationalQuantity[[U, ninf, lt, eq, gt, pinf, nan]]
  opr +[[bool ninf', bool lt', bool eq', bool gt', bool inf', bool nan']]
    (self, other: RationalQuantity[[U, ninf', lt', eq', gt', pinf', nan']]):
    RationalQuantity[[U,
      ninf ∧ (ninf' ∨ lt' ∨ eq' ∨ gt') ∨ (ninf ∨ lt ∨ eq ∨ gt) ∧ ninf',
      lt ∧ (lt' ∨ eq' ∨ gt') ∨ (lt ∨ eq ∨ gt) ∧ lt',

```

$$\begin{aligned}
& lt \wedge gt' \vee eq \wedge eq' \vee gt \wedge lt', \\
& gt \wedge (lt' \vee eq' \vee gt') \vee (lt \vee eq \vee gt) \wedge gt', \\
& pinf \wedge (lt' \vee eq' \vee gt' \vee pinf') \vee (lt \vee eq \vee gt \vee pinf) \wedge pinf', \\
& nan \vee nan' \vee ninf \wedge pinf' \vee pinf \wedge ninf'] \\
\text{opr } -(\text{self}): & \text{RationalQuantity}[[U, pinf, gt, eq, lt, ninf, nan]] \\
\text{opr } - & [[\text{bool } ninf', \text{bool } lt', \text{bool } eq', \text{bool } gt', \text{bool } pinf', \text{bool } nan']] \\
& (\text{self}, \text{other}: \text{RationalQuantity}[[U, ninf', lt', eq', gt', pinf', nan']]): \\
& \text{RationalQuantity}[[U, \\
& \quad ninf \wedge (lt' \vee eq' \vee gt' \vee pinf') \vee (ninf \vee lt \vee eq \vee gt) \wedge pinf', \\
& \quad lt \wedge (lt' \vee eq' \vee gt') \vee (lt \vee eq \vee gt) \wedge gt', \\
& \quad lt \wedge lt' \vee eq \wedge eq' \vee gt \wedge gt', \\
& \quad gt \wedge (lt' \vee eq' \vee gt') \vee (lt \vee eq \vee gt) \wedge lt', \\
& \quad pinf \wedge (ninf' \vee lt' \vee eq' \vee gt') \vee (lt \vee eq \vee gt \vee pinf) \wedge ninf', \\
& \quad nan \vee nan' \vee ninf \wedge ninf' \vee pinf \wedge pinf']] \\
\text{opr } \cdot & [[\text{unit } U', \text{bool } ninf', \text{bool } lt', \text{bool } eq', \text{bool } gt', \text{bool } pinf', \text{bool } nan']] \\
& (\text{self}, \text{other}: \text{RationalQuantity}[[U', ninf', lt', eq', gt', pinf', nan']]): \\
& \text{RationalQuantity}[[U U', \\
& \quad ninf \wedge pinf' \vee ninf \wedge gt' \vee lt \wedge pinf' \vee pinf \wedge ninf' \vee pinf \wedge lt' \vee gt \wedge ninf', \\
& \quad lt \wedge gt' \vee gt \wedge lt', \\
& \quad eq \wedge (lt' \vee eq' \vee gt') \vee (lt \vee eq \vee gt) \wedge eq', \\
& \quad lt \wedge lt' \vee gt \wedge gt', \\
& \quad ninf \wedge ninf' \vee ninf \wedge lt' \vee lt \wedge ninf' \vee pinf \wedge pinf' \vee pinf \wedge gt' \vee gt \wedge pinf', \\
& \quad nan \vee nan' \vee ninf \wedge eq' \vee pinf \wedge eq' \vee eq \wedge ninf' \vee eq \wedge pinf']] \\
\text{opr } \times & [[\text{unit } U', \text{bool } ninf', \text{bool } lt', \text{bool } eq', \text{bool } gt', \text{bool } pinf', \text{bool } nan']] \\
& (\text{self}, \text{other}: \text{RationalQuantity}[[U', ninf', lt', eq', gt', pinf', nan']]): \\
& \text{RationalQuantity}[[U U', \\
& \quad ninf \wedge pinf' \vee ninf \wedge gt' \vee lt \wedge pinf' \vee pinf \wedge ninf' \vee pinf \wedge lt' \vee gt \wedge ninf', \\
& \quad lt \wedge gt' \vee gt \wedge lt', \\
& \quad eq \wedge (lt' \vee eq' \vee gt') \vee (lt \vee eq \vee gt) \wedge eq', \\
& \quad lt \wedge lt' \vee gt \wedge gt', \\
& \quad ninf \wedge ninf' \vee ninf \wedge lt' \vee lt \wedge ninf' \vee pinf \wedge pinf' \vee pinf \wedge gt' \vee gt \wedge pinf', \\
& \quad nan \vee nan' \vee ninf \wedge eq' \vee pinf \wedge eq' \vee eq \wedge ninf' \vee eq \wedge pinf']] \\
\text{opr } /(\text{self}): & \text{RationalQuantity}[[1/U, eq, lt, ninf \vee pinf, gt, eq, nan]] \\
\text{opr } / & [[\text{unit } U', \text{bool } ninf', \text{bool } lt', \text{bool } eq', \text{bool } gt', \text{bool } pinf', \text{bool } nan']] \\
& (\text{self}, \text{other}: \text{RationalQuantity}[[U', ninf', lt', eq', gt', pinf', nan']]): \\
& \text{RationalQuantity}[[U/U', \\
& \quad ninf \wedge eq' \vee ninf \wedge gt' \vee lt \wedge eq' \vee pinf \wedge lt', \\
& \quad lt \wedge gt' \vee gt \wedge lt', \\
& \quad eq \wedge (lt' \vee gt') \vee (lt \vee eq \vee gt) \wedge (ninf' \vee pinf)', \\
& \quad lt \wedge lt' \vee gt \wedge gt', \\
& \quad ninf \wedge lt' \vee pinf \wedge eq' \vee pinf \wedge gt' \vee gt \wedge eq', \\
& \quad nan \vee nan' \vee (ninf \vee pinf) \wedge (ninf' \vee pinf') \vee eq \wedge eq']] \\
\text{opr } \lceil & \text{unit } U', \text{bool } ninf', \text{bool } lt', \text{bool } eq', \text{bool } gt', \text{bool } pinf', \text{bool } nan']] \\
& (\text{self}, \text{power}: \text{IntegerQuantity}[[U', ninf', lt', eq', gt', pinf']]): \\
& \text{RationalQuantity}[[U, \\
& \quad ninf \wedge gt' \vee eq \wedge lt', \\
& \quad lt \wedge (lt' \vee gt'), \\
& \quad eq \wedge gt' \vee (ninf \vee pinf) \wedge lt', \\
& \quad (lt \vee gt) \wedge (lt' \vee eq' \vee gt'), \\
& \quad pinf \wedge gt' \vee eq \wedge lt', \\
& \quad \text{needs more work here}] \\
\text{where } \{ & U = \text{dimensionless}, U' = \text{dimensionless} \}
\end{aligned}$$


```

opr |self|: RationalQuantity[[U, false, false, eq, lt ∨ gt, ninf ∨ pinf, nan]]
signum(self): RationalQuantity[[U, false, lt, eq, gt, false, nan]]
numerator(self): IntegerQuantity[[U, false, ninf ∨ lt, eq, gt ∨ pinf, false, nan]]
denominator(self): IntegerQuantity[[dimensionless, false, false, ninf ∨ pinf, lt ∨ eq ∨ gt, false, nan]]
floor(self): IntegerQuantity[[U, ninf, lt, eq ∨ gt, gt, pinf, nan]]
opr |self|: IntegerQuantity[[U, ninf, lt, eq ∨ gt, gt, pinf, nan]]
ceiling(self): IntegerQuantity[[U, ninf, lt, lt ∨ eq, gt, pinf, nan]]
opr [self]: IntegerQuantity[[U, ninf, lt, lt ∨ eq, gt, pinf, nan]]
round(self): IntegerQuantity[[U, ninf, lt, lt ∨ eq ∨ gt, gt, pinf, nan]]
truncate(self): IntegerQuantity[[U, ninf, lt, lt ∨ eq ∨ gt, gt, pinf, nan]]
opr [|self|]: IntegerQuantity[[U, ninf, lt, eq ∨ gt, gt, pinf, nan]] where { ¬(inf ∨ lt ∨ nan) }
opr [|self|]: IntegerQuantity[[U, ninf, lt, lt ∨ eq, gt, pinf, nan]] where { ¬(inf ∨ lt ∨ nan) }
opr [|self|]: IntegerQuantity[[U, ninf, lt, eq ∨ gt, gt, pinf, nan]] where { ¬(inf ∨ lt ∨ nan) }
opr [|self|]: IntegerQuantity[[U, ninf, lt, lt ∨ eq, gt, pinf, nan]] where { ¬(inf ∨ lt ∨ nan) }
realpart(self): RationalQuantity[[U, ninf, lt, eq, gt, pinf, nan]]
imagpart(self): RationalQuantity[[U, false, false, true, false, false, nan]]
check(self): Q throws CastException
check*(self): Q* throws CastException
check<(self): Q< throws CastException
check≤(self): Q≤ throws CastException
check≥(self): Q≥ throws CastException
check>(self): Q> throws CastException
check≠(self): Q≠ throws CastException
check*<(self): Q*< throws CastException
check*≤(self): Q*≤ throws CastException
check*≥(self): Q*≥ throws CastException
check*>(self): Q*> throws CastException
check*≠(self): Q*≠ throws CastException
check#<(self): Q#< throws CastException
check#≤(self): Q#≤ throws CastException
check#≥(self): Q#≥ throws CastException
check#>(self): Q#> throws CastException
check#≠(self): Q#≠ throws CastException
end

```

For descriptions of the methods, see Section 25.1.

38.2 The Trait `Fortress.Standard.TotalComparison`

The comparison operator `CMP`, when applied to values belonging to a total order, typically returns a value of type `TotalComparison`. The three values of type `TotalComparison` are called `LessThan`, `EqualTo`, and `GreaterThan`.

This trait supports an associative operator `LEXICO` that is useful for supporting lexicographic comparison of ordered sequences; the trick is to compare the sequences elementwise and then to use the `LEXICO` operator to reduce the sequence of comparison results. Note that `EqualTo` is the identity for `LEXICO`, and all other comparison values are left zeroes for `LEXICO`.

```
value trait TotalComparison
  extends { Comparison,
           Associative[[TotalComparison, LEXICO]],
           HasIdentity[[TotalComparison, LEXICO]],
           HasLeftZeroes[[TotalComparison, LEXICO, isLeftZeroForLEXICO]] }
  comprises { LessThan, EqualTo, GreaterThan }
  opr LEXICO(self, other: TotalComparison): TotalComparison
  isLeftZeroForLEXICO(self): Boolean
  opr ≡(self, other: TotalComparison): Boolean
  getter hashCode(): Z64
  toString(): String
end

LessThan: TotalComparison
EqualTo: TotalComparison
GreaterThan: TotalComparison
```

38.2.1 `opr LEXICO(self, other: TotalComparison): TotalComparison`

The operator `LEXICO` returns its right argument if the left argument is `EqualTo`; otherwise it returns its left argument. The `LEXICO` operator as applied to total comparison values may be described by this table:

LEXICO	LessThan	EqualTo	GreaterThan
LessThan	LessThan	LessThan	LessThan
EqualTo	LessThan	EqualTo	GreaterThan
GreaterThan	GreaterThan	GreaterThan	GreaterThan

38.2.2 `isLeftZeroForLEXICO(self): Boolean`

This method returns *false* for `EqualTo` and *true* for all other total comparison values.

38.2.3 `opr ≡(self, other: TotalComparison): Boolean`

Two total comparison values are strictly equivalent if and only if they are the same.

38.2.4 `getter hashCode():Z64`

38.2.5 `toString():String`

The `toString` method returns either “LessThan” or “EqualTo” or “GreaterThan” as appropriate.

38.3 Top-level Total Comparison Values

38.3.1 `LessThan:TotalComparison`

38.3.2 `EqualTo:TotalComparison`

38.3.3 `GreaterThan:TotalComparison`

The immutable variables `LessThan`, `EqualTo`, and `GreaterThan` have as their values the three total comparison values that respectively signify whether a left-hand comparand is less than, equal to, or greater than a right-hand comparand. They are top-level variables declared in the Fortress standard libraries.

38.4 The Trait `Fortress.Standard.Comparison`

When the comparison operator `CMP` is applied to values belonging to a partial order, rather than a total order, it typically returns a value of type `Comparison`, which includes the three values `LessThan`, `EqualTo`, and `GreaterThan` of type `TotalComparison` and also a fourth value, `Unordered`.

This trait, like trait `TotalComparison`, supports an associative operator `LEXICO` that is useful for supporting lexicographic comparison of ordered sequences; the trick is to compare the sequences elementwise and then to use the `LEXICO` operator to reduce the sequence of comparison results. Note that `EqualTo` is the identity for `LEXICO`, and all other comparison values are left zeroes for `LEXICO`.

```
value trait Comparison
  extends { IdentityEquality[[Comparison]],
            Associative[[Comparison, LEXICO]],
            HasIdentity[[Comparison, LEXICO]],
            HasLeftZeroes[[Comparison, LEXICO, isLeftZeroForLEXICO]] }
  comprises { TotalComparison, Unordered }
  opr LEXICO(self, other: Comparison): Comparison
  isLeftZeroForLEXICO(self): Boolean
  opr ≡(self, other: Comparison): Boolean
  getter hashCode():Z64
  toString():String
end
Unordered: Comparison
```

38.4.1 opr LEXICO(self, other: Comparison): Comparison

The operator LEXICO returns its right argument if the left argument is EqualTo; otherwise it returns its left argument. The LEXICO operator as applied to comparison values may be described by this table:

LEXICO	LessThan	EqualTo	GreaterThan	Unordered
LessThan	LessThan	LessThan	LessThan	LessThan
EqualTo	LessThan	EqualTo	GreaterThan	Unordered
GreaterThan	GreaterThan	GreaterThan	GreaterThan	GreaterThan
Unordered	Unordered	Unordered	Unordered	Unordered

38.4.2 isLeftZeroForLEXICO(self): Boolean

This method returns *false* for EqualTo and *true* for all other comparison values.

38.4.3 opr ≡(self, other: Comparison): Boolean

Two comparison values are strictly equivalent if and only if they are the same.

38.4.4 getter hashCode(): Z64

38.4.5 toString(): String

The *toString* method returns either “LessThan” or “EqualTo” or “GreaterThan” or “Unordered” as appropriate.

38.5 Top-level Comparison Value

38.5.1 Unordered: Comparison

The immutable variable Unordered has as its value the comparison value that signifies that two comparands are unordered. It is a top-level variable declared in the Fortress standard libraries.

Chapter 39

Components and APIs

We define a special `Fortress.Components` API that provides handles on components and APIs, and operations on them, for use by components themselves (e.g., development environments), allowing components to build and maintain other components, manipulate projects and components as objects, compile projects into components, link components together, deploy components on specific sites over the internet, etc. This API is also used by the Upgradable and Installable APIs. A component implementing this API is installed along with the Fortress standard libraries on every fortress.

Note that `Components` and `Apis` can be constructed only from the factory functions provided in the API. The components and APIs so constructed are also installed and accessible via `GetComponent`, `preferences` (which returns a list of components implementing a given API, in order of preference), and `getAPI`. Calling `preferences` on an API in the Fortress standard libraries returns a non-empty list of components. In particular, `preferences(Fortress.Components)` returns a non-empty list whose first element is the very component on which the call to `preferences` was made. Conceptually, this component serves as a handle on the enclosing fortress, which might be necessary for the purposes of certain development tools.

The operations on a fortress provided in this API take components and APIs as arguments directly, rather than names of components and APIs as the corresponding shell operations are described. This decision is done for the sake of convenience. Note, however, that a component name may be rebound on a fortress, or even uninstalled, while some process `p` keeps a reference to a corresponding `Component` object. This possibility is not problematic because the component corresponding to this object may be simply kept by the fortress until the object is freed in `p`. Also, note that `upgrade` operations on a compound component are purely functional: they produce new compound components as a result. Thus, the structure of a component as viewed through a `Component` object does not become stale in the face of upgrades.

We include a method `getSourceFile` on components that returns the source file the component was compiled from. Source files can be included with simple components during compilation as a compiler option. Doing so allows development tools such as graphical debuggers to easily display the locations of errors without the possibility that source code would not be synchronized with compiled code, as can happen in conventional programming models where compiled code is stored in nonencapsulated object files.

```
api Fortress.Components
import File from Fortress.IO
import { List, Set, Date } from Fortress.Util

trait Fortress
  fortressName: Name
  birthDate: Date
  GetComponent(componentName: Name): Component
```

```

    getAPI(apiName: Name): Api
    preferences(ofAPI: Api): <Component>
    compile(file: File): SimpleComponent
    install(file: File): Component
    install(file: File, prereqs: Set[[Api]]): Component
    upgradeAll(componentName: Name, that: Component): ()
    isValidLink(constituents: <Component>, exports = Set[[Api]], hide = Set[[Api]]): Boolean
    link(result: Name, constituents: <Component>, exports = Set[[Api]], hide = Set[[Api]]): Component
        requires { isValidLink(constituents, exports, hide) }
end
object EnclosingFortress extends { Fortress } end
trait FortressElement
    elementName: Name
    vendor: String
    owner: Fortress
    timeStamp: Date
    version: Version
    uninstall(): ()
end
trait Component extends FortressElement
    imports: Set[[Api]]
    exports: Set[[Api]]
    provides: Set[[Api]]
    visibles: Set[[Api]]
    constituents: Set[[Component]]
    run(args: String ..): ()
    constrain(destination: Name, apis: Set[[Api]]): Component
    hide(destination: Name, apis: Set[[Api]]): Component
    extract(prereqs: Set[[Api]]): File
    isValidUpgrade(that: Component): Boolean
    abstract upgrade(result: Name, that: Component): Component
        requires { self.isValidUpgrade(that) }
    sourceIsAvailable: Boolean
    getSourceFile(): File
        requires { sourceIsAvailable }
    runTests(inclusive = Boolean): ()
end
trait Api extends FortressElement
    uses: Set[[Api]]
    extraction: File
end
trait Name end
trait SimpleComponent extends Component end
trait Version
    major: N
    minor: N
end
end

```

Chapter 40

Memory Sequences and Binary Words

These are the lowest-level data structures in Fortress, upon which all others are built. Even such conceptually “primitive” data types as \mathbb{Z} and \mathbb{Z}_{32} and \mathbb{R}_{64} are defined in terms of memory sequences and binary words.

Consider, for example, the types `BinaryWord[[6]]`, `Z64`, and `N64`. All three may be regarded as 64-bit data objects. However, `Z64` causes the operator `<` to compare 64-bit words as two’s-complement signed integers, `N64` causes the operator `<` to compare 64-bit words as unsigned integers, and `BinaryWord[[6]]` does not support the operator `<` at all—instead it has two methods named *signedLT* and *unsignedLT* (which are, of course, conveniently used to implement the operator `<` for `Z64` and `N64`). Moreover, `Z64` and `N64` support units and dimensions, but `BinaryWord` values do not. The parameterized type `BinaryWord` provides methods that are only a modest abstraction of operations supported by typical hardware instruction sets and serves as the lowest-level substrate that allows types such as `Z64` to be defined by libraries coded entirely in Fortress.

Similarly, the parameterized traits `LinearSequence` and `HeapSequence` describe the lowest-level data structures that are array-like or vector-like, capable of little more than one-dimensional indexing. They serve as the lowest-level substrate that allows the complete distributed and multi-dimensional array types to be defined by libraries coded entirely in Fortress.

For convenience, we use the term *binary linear sequence* to refer to a linear sequence of binary words, and the term *binary heap sequence* to refer to a heap sequence of binary words.

```
type BinaryLinearSequence[[nat b, nat n]] = LinearSequence[[BinaryWord[[b]], n]]
type BinaryHeapSequence[[nat b]] = HeapSequence[[BinaryWord[[b]]]
```

Most operations on binary words do not depend on *endianness*, that is, in which order the bits are numbered. For operations that do depend on endianness, the parameterized trait `BinaryEndianWord` is provided.

It is also sometimes desirable to perform endianness-dependent operations on a linear sequence of binary words. For this purpose the specialized parameterized traits `BinaryLinearEndianSequence` and `BinaryEndianLinearEndianSequence` are provided; the former has `BinaryWord` values as elements, and the latter has `BinaryEndianWord` values as elements.

40.1 The Trait `Fortress.Core.LinearSequence`

A value of type `LinearSequence[[T, n]]` is a sequence of n things of type T , where n may be any natural number. Note that its length is statically fixed and may be described by a static expression. The general intent is that such a sequence will reside in a contiguous region of memory, typically belonging to a single processor or processor node, and that any element (indicated by an integer index) may be fetched or updated quickly by that processor or processor node.

If T is not a value type, then `LinearSequence[[T, n]]` describes a sequence of references, and a variable of type `LinearSequence[[T, n]]` occupies an amount of storage equal to n times the amount of storage required to hold a reference. If T is a value type, then `LinearSequence[[T, n]]` describes a sequence of “unboxed” values, and a variable of type `LinearSequence[[T, n]]` occupies an amount of storage equal to n times the amount of storage required to hold one value of type T .

Linear sequences, unlike arrays, are not too fancy. The main things you can do with linear sequences are subscripting and subscripted assignment, as well as assignment of entire sequences. They also support the concatenation operator `||`. For example:

```
x: LinearSequence[[Thread, 3]]
y: LinearSequence[[Thread, 6]]
z: LinearSequence[[Thread, 5]] = x||y[3 # 2]
```

Note in this example that the lengths are statically checkable. The range `3 # 2` is a range of constant size 2, so `y[3 # 2]` is known to be of type `LinearSequence[[Thread, 2]]`. Indeed, only ranges of static size with element type `IndexInt` may be used to subscript a linear sequence.

```
value trait LinearSequence[[T extends Any, nat n]] comprises { ... }
  coercion [[nat b, bool bigEndianSequence]
    (x: BinaryLinearEndianSequence[[b, n, bigEndianSequence]])
  where { T extends BinaryWord[[b] }
  coercion [[nat b, bool bigEndianBytes, bool bigEndianBits, bool bigEndianSequence]
    (x: BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
      n, bigEndianSequence]])
  where { T extends BinaryEndianWord[[b, bigEndianBytes, bigEndianBits] }
  coercion [[nat b, bool bigEndianBytes, bool bigEndianBits, bool bigEndianSequence]
    (x: BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
      n, bigEndianSequence]])

  where { T extends BinaryWord[[b] }
  opr [j: IndexInt]: T throws { IndexOutOfBounds }
  opr [[nat k][j: IntegerStatic[[k]]]: T where { k < n }
  opr [[nat m][r: RangeOfStaticSize[[IndexInt, m]]]: LinearSequence[[T, m]]
    throws { IndexOutOfBounds } where { m ≤ n }
  opr [[int a, nat m, int c][r: StaticRange[[a, m, c]]]: LinearSequence[[T, m]]
    where { 0 ≤ a < n, 0 ≤ a + m · c < n }
  opr [j: IndexInt] := (v: T): LinearSequence[[T, n]] throws { IndexOutOfBounds }
  opr [[nat k][j: IntegerStatic[[k]]] := (v: T): LinearSequence[[T, n]] where { k < n }
  opr [[nat m][r: RangeOfStaticSize[[IndexInt, m]]] := (v: LinearSequence[[T, m]]):
    LinearSequence[[T, n]]
    throws { IndexOutOfBounds } where { m ≤ n }
  opr [[int a, nat m, int c][r: StaticRange[[a, m, c]]] := (v: LinearSequence[[T, m]]):
    LinearSequence[[T, n]]
    where { 0 ≤ a < n, 0 ≤ a + m · c < n }
  update(j: IndexInt, v: T): LinearSequence[[T, n]] throws { IndexOutOfBounds }
```

```

update [[nat k]](j: IntegerStatic[[k]], v: T): LinearSequence[[T, n]] where { k < n }
update [[nat m]](r: RangeOfStaticSize[[IndexInt, m]], v: LinearSequence[[T, m]]):
  LinearSequence[[T, n]]
  throws { IndexOutOfBounds } where { m ≤ n }
update [[int a, nat m, int c]](r: StaticRange[[a, m, c]], v: LinearSequence[[T, m]]):
  LinearSequence[[T, n]]
  where { 0 ≤ a < n, 0 ≤ a + m · c < n }
opr || [[nat m]](self, other: LinearSequence[[T, m]]): LinearSequence[[T, n + m]]
getter reverse(): LinearSequence[[T, n]]
getter littleEndian[[nat b]](): BinaryLinearEndianSequence[[b, n, false]]
  where { T extends BinaryWord[[b]] }
getter bigEndian[[nat b]](): BinaryLinearEndianSequence[[b, n, true]]
  where { T extends BinaryWord[[b]] }
getter littleEndian[[nat b, bool bigEndianBytes, bool bigEndianBits]]():
  BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits, n, false]]
  where { T extends BinaryEndianWord[[b, bigEndianBytes, bigEndianBits]] }
getter bigEndian[[nat b, bool bigEndianBytes, bool bigEndianBits]]():
  BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits, n, true]]
  where { T extends BinaryEndianWord[[b, bigEndianBytes, bigEndianBits]] }
end

```

- 40.1.1** coercion [[nat b, bool bigEndianSequence]]
 (x: BinaryLinearEndianSequence[[b, n, bigEndianSequence]])
 where { T extends BinaryWord[[b]] }
- 40.1.2** coercion [[nat b, bool bigEndianBytes, bool bigEndianBits, bool bigEndianSequence]]
 (x: BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
 n, bigEndianSequence]])
 where { T extends BinaryEndianWord[[b, bigEndianBytes, bigEndianBits]] }

Any binary linear endian sequence may be coerced to an ordinary binary linear sequence of corresponding element type. The bit values remain the same; all that is lost is the endianness information of the sequence in the static type.

- 40.1.3** coercion [[nat b, bool bigEndianBytes, bool bigEndianBits, bool bigEndianSequence]]
 (x: BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
 n, bigEndianSequence]])
 where { T extends BinaryWord[[b]] }

A binary linear endian sequence of binary endian words may be coerced to an ordinary binary linear sequence of ordinary binary words. The bit values remain the same; all that is lost is the endianness information of both the sequence and the elements.

- 40.1.4** `opr [j: IndexInt]: T throws { IndexOutOfBoundsExpection }`
40.1.5 `opr [[nat k]][j: IntegerStatic[[k]]]: T where { k < n }`

Subscripting returns element j of this linear sequence. Indexing is zero-origin; an `IndexOutOfBoundsExpection` is thrown unless $0 \leq j < n$, where n is the length of the linear sequence. If the subscript is a static expression, then its validity is checked statically, and no exception will occur at run time.

- 40.1.6** `opr [[nat m]][r: RangeOfStaticSize[[IndexInt, m]]]: LinearSequence[[T, m]]`
`throws { IndexOutOfBoundsExpection } where { m ≤ n }`
40.1.7 `opr [[int a, nat m, int c]][r: StaticRange[[a, m, c]]]: LinearSequence[[T, m]]`
`where { 0 ≤ a < n, 0 ≤ a + m · c < n }`

Subscripting with a range of static size m returns the indicated subsequence of this linear sequence. Indexing is zero-origin; an `IndexOutOfBoundsExpection` is thrown unless $r \subseteq 0 \# n$, where n is the length of the linear sequence. If the subscript is a static range, then its validity is checked statically, and no exception will occur at run time. Element k of the result sequence is the same as element $r.lowerBound + k \times r.stride$ of this linear sequence, for all $0 \leq k < m$.

- 40.1.8** `opr [j: IndexInt] := (v: T): LinearSequence[[T, n]] throws { IndexOutOfBoundsExpection }`
40.1.9 `opr [[nat k]][j: IntegerStatic[[k]]] := (v: T): LinearSequence[[T, n]] where { k < n }`

After subscripted value object assignment, element j of the subscripted variable is the same as the given value v , and all other elements are the same as before. Indexing is zero-origin; an `IndexOutOfBoundsExpection` is thrown unless $0 \leq j < n$, where n is the length of the linear sequence. If the subscript is a static expression, then its validity is checked statically, and no exception will occur at run time.

- 40.1.10** `opr [[nat m]][r: RangeOfStaticSize[[IndexInt, m]]] := (v: LinearSequence[[T, m]]):`
`LinearSequence[[T, n]]`
`throws { IndexOutOfBoundsExpection } where { m ≤ n }`
40.1.11 `opr [[int a, nat m, int c]][r: StaticRange[[a, m, c]]] := (v: LinearSequence[[T, m]]):`
`LinearSequence[[T, n]]`
`where { 0 ≤ a < n, 0 ≤ a + m · c < n }`

After subscripted value object assignment, elements of the subscripted variable selected by r are the same as corresponding elements of v , and all other elements are the same as before; specifically, element $r.lowerBound + k \times r.stride$ of the updated variable is the same as element k of v , for all $0 \leq k < m$. Indexing is zero-origin; an `IndexOutOfBoundsExpection` is thrown unless $r \subseteq 0 \# n$, where n is the length of the linear sequence. If the subscript is a static range, then its validity is checked statically, and no exception will occur at run time.

- 40.1.12** `update(j: IndexInt, v: T): LinearSequence[[T, n]] throws { IndexOutOfBoundsException }`
40.1.13 `update[[nat k]](j: IntegerStatic[[k]], v: T): LinearSequence[[T, n]] where { k < n }`

This is a functional version of subscripted value object assignment: element j of the result is the same as the given value v , and all other elements are the same as before. Indexing is zero-origin; an `IndexOutOfBoundsException` is thrown unless $0 \leq j < n$, where n is the length of the linear sequence. If the subscript is a static expression, then its validity is checked statically, and no exception will occur at run time.

- 40.1.14** `update[[nat m]](r: RangeOfStaticSize[[IndexInt, m]], v: LinearSequence[[T, m]]):
LinearSequence[[T, n]]
throws { IndexOutOfBoundsException } where { m ≤ n }`
40.1.15 `update[[int a, nat m, int c]](r: StaticRange[[a, m, c]], v: LinearSequence[[T, m]]):
LinearSequence[[T, n]]
where { 0 ≤ a < n, 0 ≤ a + m · c < n }`

This is a functional version of subscripted value object assignment: elements of the result selected by r are the same as corresponding elements of v , and all other elements are the same as before; specifically, element $r.lowerBound + k \times r.stride$ of the result is the same as element k of v , for all $0 \leq k < m$. Indexing is zero-origin; an `IndexOutOfBoundsException` is thrown unless $0 \leq j < n$, where n is the length of the linear sequence. If the subscript is a static range, then its validity is checked statically, and no exception will occur at run time.

- 40.1.16** `opr || [[nat m]](self, other: LinearSequence[[T, m]]): LinearSequence[[T, n + m]]`

The result is a linear sequence whose length is equal to the sum of the lengths of this linear sequence and the other linear sequence. Element k of the result is the same as element k of this linear sequence if $0 \leq k < n$, and is the same as element $k - n$ of the other linear sequence if $n \leq k < n + m$.

- 40.1.17** `getter reverse(): LinearSequence[[T, n]]`

The result is a linear sequence such that element k of the result is the same as element $n - 1 - k$ of this linear sequence, for all $0 \leq k < n$.

- 40.1.18** `getter littleEndian[[nat b]](): BinaryLinearEndianSequence[[b, n, false]]
where { T extends BinaryWord[[b]] }`
40.1.19 `getter bigEndian[[nat b]](): BinaryLinearEndianSequence[[b, n, true]]
where { T extends BinaryWord[[b]] }`
40.1.20 `getter littleEndian[[nat b, bool bigEndianBytes, bool bigEndianBits]]():
BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits, n, false]]
where { T extends BinaryEndianWord[[b, bigEndianBytes, bigEndianBits]] }`
40.1.21 `getter bigEndian[[nat b, bool bigEndianBytes, bool bigEndianBits]]():
BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits, n, true]]
where { T extends BinaryEndianWord[[b, bigEndianBytes, bigEndianBits]] }`

These conversion getters allow a linear sequence of (possibly endian) binary words to be treated as a specifically little-endian or specifically big-endian linear sequence. This is especially useful just before invoking an endianness-dependent method, for example `s.littleEndian.countLeadingZeroBits()`.

40.2 Constructing Linear Sequences

40.2.1 `makeLinearSequence`[[*T* extends Any, nat *n*]](*item*:*T*): LinearSequence[[*T*, *n*]]

A new linear sequence of length *n* is returned. Every element of the linear sequence is initialized to be the same as the given *item*.

40.2.2 `computeLinearSequence`[[*T* extends Any, nat *n*]](*f*: IndexInt → *T*): LinearSequence[[*T*, *n*]]

A new linear sequence of length *n* is returned. Element *j* of the new linear sequence is initialized to a value computed by calling the given function *f* with argument *j*.

40.3 The Trait `Fortress.Core.HeapSequence`

A value of type `HeapSequence`[[*T*]] is an array-like object that contains items of type *T*. The length of a heap sequence is in general not known statically, but can be discovered by asking for its length. A variable of type `HeapSequence`[[*T*]] occupies the amount of storage required to hold a reference; this reference refers to an object that occupies an amount of storage greater than or equal to the amount that would be occupied by a variable of type `LinearSequence`[[*T*, *n*]] where *n* is the length of the heap sequence.

Heap sequences, like linear sequences and unlike arrays, are not too fancy. The main things you can do with heap sequences are subscripting and subscripted assignment. Concatenation is *not* supported, because a basic principle of the low-level types is that none of the operations, other than explicit construction of a heap sequence, does any heap allocation. However, a range of static size may be used to index a heap sequence; the result is a linear sequence.

```
trait HeapSequence[[T extends Any]] comprises { ... }
  opr [j: IndexInt]: T throws { IndexOutOfBoundsException }
  opr [[nat m][r: RangeOfStaticSize[[IndexInt, m]]]: LinearSequence[[T, m]]
    throws { IndexOutOfBoundsException }
  opr [j: IndexInt] := (v: T): () throws { IndexOutOfBoundsException }
  opr [[nat m][r: RangeOfStaticSize[[IndexInt, m]]] := (v: LinearSequence[[T, m]]): ()
    throws { IndexOutOfBoundsException }
  reverse(selfStart: IndexInt, length: IndexInt): () throws { IndexOutOfBoundsException }
  opr |self| : IndexInt
end
```

40.3.1 `opr` [*j*: IndexInt]: *T* throws { IndexOutOfBoundsException }

Subscripting returns element *j* of this heap sequence. Indexing is zero-origin; an `IndexOutOfBoundsException` is thrown unless $0 \leq j < n$, where *n* is the length of the heap sequence.

40.3.2 `opr` $\llbracket \text{nat } m \rrbracket [r: \text{RangeOfStaticSize} \llbracket \text{IndexInt}, m \rrbracket]: \text{LinearSequence} \llbracket T, m \rrbracket$
`throws` { `IndexOutOfBounds`Exception }

Subscripting by a range returns the indicated subsequence of this heap sequence. The range must be a range of static size, and the result is returned as a linear sequence (not a heap sequence), so no heap allocation is performed. Indexing is zero-origin; an `IndexOutOfBounds`Exception is thrown unless $r \subseteq (0 : n - 1)$, where n is the length of the heap sequence. Element k of the result linear sequence is the same as element $r.lowerBound + k \times r.stride$ of this heap sequence, for all $0 \leq k < m$.

40.3.3 `opr` $[j: \text{IndexInt}] := (v: T): ()$ `throws` { `IndexOutOfBounds`Exception }

After subscripted assignment, element j of this heap sequence is the same as the given value v , and all other elements are the same as before. Indexing is zero-origin; an `IndexOutOfBounds`Exception is thrown unless $0 \leq j < n$, where n is the length of the heap sequence.

40.3.4 `opr` $\llbracket \text{nat } m \rrbracket [r: \text{RangeOfStaticSize} \llbracket \text{IndexInt}, m \rrbracket] := (v: \text{LinearSequence} \llbracket T, m \rrbracket): ()$
`throws` { `IndexOutOfBounds`Exception }

After subscripted assignment using a range as a subscript, elements of the subscripted variable selected by r are the same as corresponding elements of v , and all other elements are the same as before; specifically, element $r.lowerBound + k \times r.stride$ of the updated variable is the same as element k of v , for all $0 \leq k < m$. The range must be a range of static size, and the values to be assigned must be passed as linear sequence of the same size. Indexing is zero-origin; an `IndexOutOfBounds`Exception is thrown unless $r \subseteq (0 : n - 1)$, where n is the length of the heap sequence.

40.3.5 `reverse`($selfStart: \text{IndexInt}, length: \text{IndexInt}$): () `throws` { `IndexOutOfBounds`Exception }

Elements $selfStart$ through $selfStart + length - 1$, inclusive, are reversed in order, that is, rearranged so that the value originally stored at element $selfStart + j$ becomes element $selfStart + length - 1 - j$, for all $0 \leq j < length$. Other elements of this heap sequence are unaffected.

40.3.6 `opr` $|self|: \text{IndexInt}$

The length of this heap sequence is returned. Note that the size of a heap sequence is specified at run time when the heap sequence is created; once a heap sequence has been created, its length does not change.

40.4 Constructing Heap Sequences

40.4.1 *makeHeapSequence*[[*T* extends Any]](*n*: IndexInt, *item*: *T*): HeapSequence[[*T*]]
throws { NegativeLengthException }

A new heap sequence of length *n* is allocated and returned. A `NegativeLengthException` is thrown if $n < 0$. Every element of the heap sequence is initialized to be the same as the given *item*.

40.4.2 *computeHeapSequence*[[*T* extends Any]](*n*: IndexInt, *f*: IndexInt → *T*): HeapSequence[[*T*]]
throws { NegativeLengthException }

A new heap sequence of length *n* is allocated and returned. A `NegativeLengthException` is thrown if $n < 0$. Element *j* of the new heap sequence is initialized to a value computed by calling the given function *f* with argument *j*.

40.5 The Trait Fortress.Core.BinaryWord

A value of type `BinaryWord[[b]]` is a binary word of 2^b bits; b may be any natural number, so `BinaryWord[[0]]` is a bit, `BinaryWord[[3]]` is a byte, `BinaryWord[[6]]` is a 64-bit word, and `BinaryWord[[10]]` is a 1024-bit word. In fact, for convenience, the type abbreviations `Bit` and `Byte` are defined:

```
type Bit = BinaryWord[[0]]
type Byte = BinaryWord[[3]]
```

The type `BinaryWord[[b]]` has $2^{(2^b)}$ distinct values. When the binary word is regarded as an unsigned integer, these values are identified with the nonnegative integers that are less than $2^{(2^b)}$. A binary word may also be regarded as a signed integer: a value that, when regarded as an unsigned integer, is identified with an integer less than $2^{(2^b-1)}$, is identified with that same integer when regarded as a signed integer; but a value that, when regarded as an unsigned integer, is identified with an integer not less than $2^{(2^b-1)}$, is identified with that same integer less $2^{(2^b)}$. (This is the standard “two’s complement” representation for signed integers.)

A binary word of one bit can have one of two values, 0 or 1. A binary word of more than one bit has two halves, a high half and a low half, which are binary words of half the size. If v is the unsigned integer value of a binary word of 2^b bits, $b \geq 1$, h is the unsigned integer value of its high half, and l is the unsigned integer value of its low half, then $v = h \cdot 2^{2^{b-1}} + l$.

Operations on binary words include bitwise boolean operations, arithmetic operations, shifts and rotates, population count, and counting of leading and trailing zeros. The type `BinaryWord[[b]]` is not “endian” and has no operations that depend on endianness. However, if w is binary word, then $w.*littleEndian*$ is a little-endian version of w and $w.*bigEndian*$ is a big-endian version of w ; for example, if w is of type `BinaryWord[[6]]`, then $w.*littleEndian*_{63}$ is the most significant bit (the sign bit if the word is regarded as a two’s-complement integer), and $w.*bigEndian*_0$ is that same bit.

```
trait BinaryWord[[nat b]] extends { BasicBinaryWordOperations[[BinaryWord[[b]]] }
  comprises { ... }
  where { b ≤ maxBinaryWordBitLog }
  coercion [[int r]](x: IntegerStatic[[r]]) where { -2^{b-1} ≤ r < 2^b }
  coercion [[bool bigEndianBytes, bool bigEndianBits]]
    (x: BinaryEndianWord[[b, bigEndianBytes, bigEndianBits]])
  coercion [[nat b', nat n, bool bigEndianSequence]]
    (x: BinaryLinearEndianSequence[[b', n, bigEndianSequence]])
  where { 2^b = n · 2^{b'} }
  coercion [[nat b', bool bigEndianBytes, bool bigEndianBits, nat n, bool bigEndianSequence]]
    (x: BinaryEndianLinearEndianSequence[[b', bigEndianBytes, bigEndianBits,
      n, bigEndianSequence]])
  where { 2^b = n · 2^{b'} }
  bit(m: IndexInt): Bit
  getter lowHalf(): BinaryWord[[b - 1]] where { b > 0 }
  getter highHalf(): BinaryWord[[b - 1]] where { b > 0 }
  opr || [[nat m]](self, other: BinaryWord[[b]]): BinaryWord[[b + 1]]
  where { b < maxBinaryWordBitLog }
  bitShuffle(other: BinaryWord[[b]]): BinaryWord[[b + 1]]
  where { b < maxBinaryWordBitLog }
  bitUnshuffle(): (BinaryWord[[b - 1]], BinaryWord[[b - 1]]) where { b > 0 }
end
```

40.5.1 coercion $\llbracket \text{int } r \rrbracket (x: \text{IntegerStatic} \llbracket r \rrbracket)$ where $\{-2^{b-1} \leq r < 2^b\}$

An static integer may be coerced to a binary word that corresponds to that integer value when interpreted as either a signed integer or an unsigned integer. For example, the type `BinaryWord[3]` has $2^{(2^3)} = 256$ distinct binary word values; when they are interpreted as signed integers, the integer values range from -128 to 127 , and when they are interpreted as unsigned integers, the integer values range from 0 to 255 . Therefore any static integer from -128 to 255 may be coerced to type `BinaryWord[3]`.

40.5.2 coercion $\llbracket \text{bool } \text{bigEndianBytes}, \text{bool } \text{bigEndianBits} \rrbracket$
 $(x: \text{BinaryEndianWord} \llbracket b, \text{bigEndianBytes}, \text{bigEndianBits} \rrbracket)$

Any binary endian word may be coerced to a plain binary word of the same size and value. In effect, this coercion merely discards the endianness information.

40.5.3 coercion $\llbracket \text{nat } b', \text{nat } n, \text{bool } \text{bigEndianSequence} \rrbracket$
 $(x: \text{BinaryLinearEndianSequence} \llbracket b', n, \text{bigEndianSequence} \rrbracket)$
where $\{2^b = n \cdot 2^{b'}\}$

A binary linear endian sequence of smaller binary words may be coerced to a single binary word, provided that the length of the linear sequence is an appropriate power of two, so that the total number of bits in the sequence is the same as the total number of bits in the resulting binary word. The manner in which the elements of the sequence are used to form the new binary word value respects the endianness of the sequence, so that element 0 of the sequence supplies the most significant bits of the result if `bigEndianSequence` is true, but supplies the least significant bits of the result if `bigEndianSequence` is false.

40.5.4 coercion $\llbracket \text{nat } b', \text{bool } \text{bigEndianBytes}, \text{bool } \text{bigEndianBits}, \text{nat } n, \text{bool } \text{bigEndianSequence} \rrbracket$
 $(x: \text{BinaryEndianLinearEndianSequence} \llbracket b', \text{bigEndianBytes}, \text{bigEndianBits},$
 $n, \text{bigEndianSequence} \rrbracket)$
where $\{2^b = n \cdot 2^{b'}\}$

A binary endian linear endian sequence of smaller binary endian words may be coerced to a single binary word, provided that the length of the linear sequence is an appropriate power of two, so that the total number of bits in the sequence is the same as the total number of bits in the resulting binary word. The manner in which the elements of the sequence are used to form the new binary word value respects the endianness of the sequence, so that element 0 of the sequence supplies the most significant bits of the result if `bigEndianSequence` is true, but supplies the least significant bits of the result if `bigEndianSequence` is false. In effect, the “bytes and bits” endianness information is simply ignored and discarded.

40.5.5 *bit*(*m*: IndexInt): Bit

The result is a bit whose value (0 or 1) is equal to $\lfloor v \cdot 2^{-m} \rfloor \bmod 2$ where v is the value of the binary word regarded as an unsigned integer. This formula holds for any value of m ; note that if m is negative or greater than $2^b - 1$, the result will always be a 0-bit. Thus the *bit* method provides a kind of “little-endian” indexing of the bits of a binary word, even a binary word whose type is not intrinsically endian, but it does not require that the bit number identify an actual represented bit of the binary word.

The *bit* method is particularly useful for describing the behavioral properties of other methods of binary data.

40.5.6 getter *lowHalf*(): BinaryWord[[*b* - 1]] where { *b* > 0 }

40.5.7 getter *highHalf*(): BinaryWord[[*b* - 1]] where { *b* > 0 }

The getters *lowHalf* and *highHalf* each return a binary word of half the size (in bits) of this binary word; *lowHalf* returns the less significant bits, and *highHalf* returns the more significant bits.

$$\begin{aligned} \text{property } \forall(v) \quad & \bigwedge_{m \leftarrow 0 \# 2^{b-1}} v.\text{lowHalf}.\text{bit}(m) = v.\text{bit}(m) \\ \text{property } \forall(v) \quad & \bigwedge_{m \leftarrow 0 \# 2^{b-1}} v.\text{highHalf}.\text{bit}(m) = v.\text{bit}(m + 2^{b-1}) \end{aligned}$$

40.5.8 opr || [[nat *m*]](*self*, *other*: BinaryWord[[*b*]]): BinaryWord[[*b* + 1]] where { *b* < *maxBinaryWordBitLog* }

The result of concatenating two binary words of size 2^b is a single binary word of size 2^{b+1} . The left-hand operand becomes the high (more significant) half of the result and the right-hand operand becomes the low (less significant) half of the result.

$$\begin{aligned} \text{property } \forall(v, w) \quad & (v \parallel w).\text{highHalf} = v \\ \text{property } \forall(v, w) \quad & (v \parallel w).\text{lowHalf} = w \end{aligned}$$

40.5.9 *bitShuffle*(*other*: BinaryWord[[*b*]]): BinaryWord[[*b* + 1]] where { *b* < *maxBinaryWordBitLog* }

The bit-shuffle operation interleaves the bits of two words, as if shuffling cards together (using what magicians call a “perfect shuffle”). The result of shuffling the bits two binary words of size 2^b is a single binary word of size 2^{b+1} . This binary word provides the odd-numbered bits of the result and the other binary word provides the even-numbered bits of the result. For example, shuffling 1111 and 0000 produces 10101010.

$$\text{property } \forall(v, w, m: \text{IndexInt}) \quad v.\text{bitShuffle}(w).\text{bit}(m) = (\text{if } \text{odd } m \text{ then } v.\text{bit}((m - 1)/2) \text{ else } w.\text{bit}(m/2))$$

40.5.10 *bitUnshuffle()*: (BinaryWord $\llbracket b - 1 \rrbracket$, BinaryWord $\llbracket b - 1 \rrbracket$) where $\{ b > 0 \}$

This is the inverse of the *bitShuffle* method: the odd-numbered bits of this binary word are used to form a binary word of half the size, and likewise the even-numbered bits, and a tuple of the two binary words is returned.

property $\forall (v, w) v.\text{bitShuffle}(w).\text{bitUnshuffle}() = (v, w)$

40.6 The Trait Fortress.Core.BinaryEndianWord

The type `BinaryEndianWord[b, bigEndianBytes, bigEndianBits]` is exactly like `BinaryWord[b]` but bears two kinds of endianness information. A `BinaryEndianWord` may be split into a sequence of smaller words; the result is of type `BinaryLinearEndianSequence`. The flag `bigEndianBytes` indicates whether subword 0 is the most significant subword (if `bigEndianBytes` is true) or least significant subword (if `bigEndianBytes` is false) of the original binary word. A `BinaryEndianWord` may also be subscripted to extract a bit or a bit field; the flag `bigEndianBits` indicates whether bit 0 is the most significant bit (if `bigEndianBits` is true) or least significant bit (if `bigEndianBits` is false) of the original binary word. (Yes, it may seem strange for the bit ordering to differ from the subword ordering, but they do differ on a number of architectures, including SPARC.) Extracting a bit produces a `Bit`, that is, a `BinaryWord[0]`. Extracting a bit field of width k produces a `BinaryEndianLinearEndianSequence` with $n = k$ and $b = 0$; the endianness of the sequence matches the bit-endianness of the original `BinaryEndianWord`.

```

trait BinaryEndianWord[nat b, bool bigEndianBytes, bool bigEndianBits]
  extends { BasicBinaryWordOperations[BinaryEndianWord[b, bigEndianBytes, bigEndianBits], b] }
  comprises { ... }
  where { b ≤ maxBinaryWordBitLog }
  coercion [[int r](x: IntegerStatic[r]) where { -2b-1 ≤ r < 2b }
opr [j: IndexInt] : BinaryEndianWord[1, bigEndianBytes, bigEndianBits]
  throws { IndexOutOfBounds }
opr [[nat k][j: IntegerStatic[k]] : BinaryEndianWord[1, bigEndianBytes, bigEndianBits]
  where { k < 2b }
opr [[nat m][r: RangeOfStaticSize[IndexInt, m]] :
  BinaryEndianLinearEndianSequence[1, bigEndianBytes, bigEndianBits, m, bigEndianBits]
  throws { IndexOutOfBounds }
opr [[int a, nat m, int c][r: StaticRange[a, m, c]] :
  BinaryEndianLinearEndianSequence[1, bigEndianBytes, bigEndianBits, m, bigEndianBits]
  where { 0 ≤ a < 2b, 0 ≤ a + m · c < 2b }
opr [j: IndexInt] := (v: Bit):
  BinaryEndianWord[b, bigEndianBytes, bigEndianBits]
  throws { IndexOutOfBounds }
opr [[nat k][j: IntegerStatic[k]] := (v: Bit):
  BinaryEndianWord[b, bigEndianBytes, bigEndianBits]
  where { k < 2b }
opr [[nat m][r: RangeOfStaticSize[IndexInt, m]] := (v: BinaryLinearSequence[1, m]):
  BinaryEndianWord[b, bigEndianBytes, bigEndianBits]
  throws { IndexOutOfBounds }
opr [[int a, nat m, int c][r: StaticRange[a, m, c]] := (v: BinaryLinearSequence[1, k]):
  BinaryEndianWord[b, bigEndianBytes, bigEndianBits]
  where { 0 ≤ a < 2b, 0 ≤ a + m · c < 2b }
update(j: IndexInt, v: Bit):
  BinaryEndianWord[b, bigEndianBytes, bigEndianBits]
  throws { IndexOutOfBounds }
update[[nat k](j: IntegerStatic[k], v: Bit):
  BinaryEndianWord[b, bigEndianBytes, bigEndianBits]
  where { k < 2b }
update[[nat m](r: RangeOfStaticSize[IndexInt, m], v: BinaryLinearSequence[1, m]):
  BinaryEndianWord[b, bigEndianBytes, bigEndianBits]
  throws { IndexOutOfBounds }
update[[int a, nat m, int c](r: StaticRange[a, m, c], v: BinaryLinearSequence[1, m]):
  BinaryEndianWord[b, bigEndianBytes, bigEndianBits]
  where { 0 ≤ a < 2b, 0 ≤ a + m · c < 2b }

```

```

lowHalf(): BinaryEndianWord[[b - 1, bigEndianBytes, bigEndianBits]] where { b > 0 }
highHalf(): BinaryEndianWord[[b - 1, bigEndianBytes, bigEndianBits]] where { b > 0 }
opr || [[nat m, bool bigEndianSequence]
  (self, other: BinaryEndianWord[[b, bigEndianBytes, bigEndianBits]]):
    BinaryEndianLinearEndianSequence[[b + 1, bigEndianBytes, bigEndianBits,
      2, bigEndianSequence]]
  where { bigEndianSequence = bigEndianBytes }
opr || [[nat m, bool bigEndianSequence, nat radix, nat q, nat k, nat v]
  (self, other: NaturalNumeral[[m, radix, v]]):
    BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
      k + 1, bigEndianSequence]]
  where { bigEndianSequence = bigEndianBytes, radix = 2q, q · m = k · 2b }
opr || [[nat m, bool bigEndianSequence, nat radix, nat q, nat k, nat v]
  (other: NaturalNumeral[[m, radix, v]], self):
    BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
      k + 1, bigEndianSequence]]
  where { bigEndianSequence = bigEndianBytes, radix = 2q, q · m = k · 2b }
bitShuffle(other: BinaryEndianWord[[b, bigEndianBytes, bigEndianBits]]):
  BinaryEndianWord[[b + 1, bigEndianBytes, bigEndianBits]]
  where { b < maxBinaryWordBitLog }
bitUnshuffle(): (BinaryEndianWord[[b - 1, bigEndianBytes, bigEndianBits]],
  BinaryEndianWord[[b - 1, bigEndianBytes, bigEndianBits]])
  where { b > 0 }
littleEndian(): BinaryEndianWord[[b, false, false]]
bigEndian(): BinaryEndianWord[[b, true, true]]
end

```

40.6.1 coercion [[int r]](x: IntegerStatic[[r]]) where { $-2^{b-1} \leq r < 2^b$ }

An static integer may be coerced to a binary endian word exactly as if it were coerced to a plain binary word of the same size; the endian numbering of the bytes and bits does not affect which binary word value is produced from the static integer.

40.6.2 opr [j: IndexInt] : BinaryEndianWord[[1, bigEndianBytes, bigEndianBits]]
throws { IndexOutOfBoundsException }

40.6.3 opr [[nat k]] [j: IntegerStatic[[k]]] : BinaryEndianWord[[1, bigEndianBytes, bigEndianBits]]
where { $k < 2^b$ }

Subscripting returns bit j of this binary endian word. The numbering of the bits is dictated by *bigEndianBits*. Indexing is zero-origin; an IndexOutOfBoundsException is thrown unless $0 \leq j < 2^b$. If the subscript is a static expression, then its validity is checked statically, and no exception will occur at run time.

property $\forall(v) \bigwedge_{j \leftarrow 0 \# 2^b} v_j = (\text{if } \text{bigEndianBits} \text{ then } v.\text{bit}(2^b - 1 - j) \text{ else } v.\text{bit}(j) \text{ end})$

40.6.4 `opr` $\llbracket \text{nat } m \rrbracket [r: \text{RangeOfStaticSize} \llbracket \text{IndexInt}, m \rrbracket] :$
`BinaryEndianLinearEndianSequence` $\llbracket 1, \text{bigEndianBytes}, \text{bigEndianBits}, m, \text{bigEndianBits} \rrbracket$
`throws` { `IndexOutOfBoundsException` }

40.6.5 `opr` $\llbracket \text{int } a, \text{nat } m, \text{int } c \rrbracket [r: \text{StaticRange} \llbracket a, m, c \rrbracket] :$
`BinaryEndianLinearEndianSequence` $\llbracket 1, \text{bigEndianBytes}, \text{bigEndianBits}, m, \text{bigEndianBits} \rrbracket$
`where` { $0 \leq a < 2^b, 0 \leq a + m \cdot c < 2^b$ }

Subscripting with a range of static size m returns the indicated subsequence of bits of this binary endian word. The numbering of the bits is dictated by *bigEndianBits*. The result is a binary endian linear endian sequence of bits whose sequence endianness is the same as the bit endianness of this binary endian word. Indexing is zero-origin; an `IndexOutOfBoundsException` is thrown unless $r \subseteq 0 \# 2^b$. If the subscript is a static range, then its validity is checked statically, and no exception will occur at run time. Element k of the result sequence is the same as the bit that would be selected from this binary endian word by subscripting it with $r.lowerBound + k \times r.stride$, for all $0 \leq k < m$.

40.6.6 `opr` $[j: \text{IndexInt}] := (v: \text{Bit}) :$
`BinaryEndianWord` $\llbracket b, \text{bigEndianBytes}, \text{bigEndianBits} \rrbracket$
`throws` { `IndexOutOfBoundsException` }

40.6.7 `opr` $\llbracket \text{nat } k \rrbracket [j: \text{IntegerStatic} \llbracket k \rrbracket] := (v: \text{Bit}) :$
`BinaryEndianWord` $\llbracket b, \text{bigEndianBytes}, \text{bigEndianBits} \rrbracket$
`where` { $k < 2^b$ }

After subscripted value object assignment, the bit that would be selected from this binary endian word by subscripting it with j is the same as the given bit v , and all other bits are the same as before. Indexing is zero-origin; an `IndexOutOfBoundsException` is thrown unless $0 \leq j < 2^b$. If the subscript is a static expression, then its validity is checked statically, and no exception will occur at run time.

40.6.8 `opr` $\llbracket \text{nat } m \rrbracket [r: \text{RangeOfStaticSize} \llbracket \text{IndexInt}, m \rrbracket] := (v: \text{BinaryLinearSequence} \llbracket 1, m \rrbracket) :$
`BinaryEndianWord` $\llbracket b, \text{bigEndianBytes}, \text{bigEndianBits} \rrbracket$
`throws` { `IndexOutOfBoundsException` }

40.6.9 `opr` $\llbracket \text{int } a, \text{nat } m, \text{int } c \rrbracket [r: \text{StaticRange} \llbracket a, m, c \rrbracket] := (v: \text{BinaryLinearSequence} \llbracket 1, m \rrbracket) :$
`BinaryEndianWord` $\llbracket b, \text{bigEndianBytes}, \text{bigEndianBits} \rrbracket$
`where` { $0 \leq a < 2^b, 0 \leq a + m \cdot c < 2^b$ }

After subscripted value object assignment, bits that would be selected from this binary endian word by subscripting it with r are the same as corresponding elements of v , and all other bits are the same as before; specifically, the bit that would be selected from this binary endian word by subscripting it with $r.lowerBound + k \times r.stride$ is the same as element k of v , for all $0 \leq k < m$. Indexing is zero-origin; an `IndexOutOfBoundsException` is thrown unless $r \subseteq 0 \# 2^b$. If the subscript is a static range, then its validity is checked statically, and no exception will occur at run time.

40.6.10 *update*(*j*: IndexInt, *v*: Bit):
 BinaryEndianWord[[*b*, *bigEndianBytes*, *bigEndianBits*]]
 throws { IndexOutOfBoundsException }

40.6.11 *update*[[nat *k*]](*j*: IntegerStatic[[*k*]], *v*: Bit):
 BinaryEndianWord[[*b*, *bigEndianBytes*, *bigEndianBits*]]
 where { *k* < 2^{*b*} }

This is a functional version of subscripted value object assignment: the bit that would be selected from the result by subscripting it with *j* is the same as the given bit *v*, and all other bits are the same as before. Indexing is zero-origin; an IndexOutOfBoundsException is thrown unless $0 \leq j < 2^b$. If the subscript is a static expression, then its validity is checked statically, and no exception will occur at run time.

40.6.12 *update*[[nat *m*]](*r*: RangeOfStaticSize[[IndexInt, *m*]], *v*: BinaryLinearSequence[[1, *m*]]):
 BinaryEndianWord[[*b*, *bigEndianBytes*, *bigEndianBits*]]
 throws { IndexOutOfBoundsException }

40.6.13 *update*[[int *a*, nat *m*, int *c*]](*r*: StaticRange[[*a*, *m*, *c*]], *v*: BinaryLinearSequence[[1, *m*]]):
 BinaryEndianWord[[*b*, *bigEndianBytes*, *bigEndianBits*]]
 where { $0 \leq a < 2^b, 0 \leq a + m \cdot c < 2^b$ }

This is a functional version of subscripted value object assignment: bits that would be selected from the result by subscripting it with *r* are the same as corresponding elements of *v*, and all other bits are the same as before; specifically, the bit that would be selected from the result by subscripting it with *r.lowerBound* + *k* × *r.stride* is the same as element *k* of *v*, for all $0 \leq k < m$. Indexing is zero-origin; an IndexOutOfBoundsException is thrown unless $r \subseteq 0 \# 2^b$. If the subscript is a static range, then its validity is checked statically, and no exception will occur at run time.

40.6.14 *lowHalf*(): BinaryEndianWord[[*b* - 1, *bigEndianBytes*, *bigEndianBits*]] where { *b* > 0 }

40.6.15 *highHalf*(): BinaryEndianWord[[*b* - 1, *bigEndianBytes*, *bigEndianBits*]] where { *b* > 0 }

The getters *lowHalf* and *highHalf* each return a binary endian word of half the size (in bits) of this binary endian word, and with the same endian characteristics; *lowHalf* returns the less significant bits, and *highHalf* returns the more significant bits.

property $\forall(v) \bigwedge_{m \leftarrow 0 \# 2^{b-1}} v.\text{lowHalf}.\text{bit}(m) = v.\text{bit}(m)$

property $\forall(v) \bigwedge_{m \leftarrow 0 \# 2^{b-1}} v.\text{highHalf}.\text{bit}(m) = v.\text{bit}(m + 2^{b-1})$

property $\forall(v) \bigwedge_{m \leftarrow 0 \# 2^{b-1}} v.\text{lowHalf}_m = v[\text{if } \text{bigEndianBits} \text{ then } m + 2^{b-1} \text{ else } m \text{ end}]$

property $\forall(v) \bigwedge_{m \leftarrow 0 \# 2^{b-1}} v.\text{highHalf}_m = v[\text{if } \text{bigEndianBits} \text{ then } m \text{ else } m + 2^{b-1} \text{ end}]$

40.6.16 `opr` || `[[nat m, bool bigEndianSequence]]`
 (`self, other`: BinaryEndianWord`[[b, bigEndianBytes, bigEndianBits]]`):
 BinaryEndianLinearEndianSequence`[[b + 1, bigEndianBytes, bigEndianBits,`
`2, bigEndianSequence]]`
 where { `bigEndianSequence = bigEndianBytes` }

[Description to be supplied.]

40.6.17 `opr` || `[[nat m, bool bigEndianSequence, nat radix, nat q, nat k, nat v]]`
 (`self, other`: NaturalNumeral`[[m, radix, v]]`):
 BinaryEndianLinearEndianSequence`[[b, bigEndianBytes, bigEndianBits,`
`k + 1, bigEndianSequence]]`
 where { `bigEndianSequence = bigEndianBytes, radix = 2q, q · m = k · 2b` }

[Description to be supplied.]

40.6.18 `opr` || `[[nat m, bool bigEndianSequence, nat radix, nat q, nat k, nat v]]`
 (`other`: NaturalNumeral`[[m, radix, v]]`, `self`):
 BinaryEndianLinearEndianSequence`[[b, bigEndianBytes, bigEndianBits,`
`k + 1, bigEndianSequence]]`
 where { `bigEndianSequence = bigEndianBytes, radix = 2q, q · m = k · 2b` }

[Description to be supplied.]

40.6.19 `bitShuffle`(`other`: BinaryEndianWord`[[b, bigEndianBytes, bigEndianBits]]`):
 BinaryEndianWord`[[b + 1, bigEndianBytes, bigEndianBits]]`
 where { `b < maxBinaryWordBitLog` }

[Description to be supplied.]

40.6.20 `bitUnshuffle`(): (BinaryEndianWord`[[b - 1, bigEndianBytes, bigEndianBits]]`,
 BinaryEndianWord`[[b - 1, bigEndianBytes, bigEndianBits]]`)
 where { `b > 0` }

[Description to be supplied.]

40.6.21 `littleEndian`(): BinaryEndianWord`[[b, false, false]]`

40.6.22 `bigEndian`(): BinaryEndianWord`[[b, true, true]]`

[Description to be supplied.]

40.7 The Trait Fortress.Core.BasicBinaryOperations

```
trait BasicBinaryOperations[[T extends BasicBinaryOperations[[T]]]
  wrappingAdd(other: T): T
  add(other: T, carryIn: Bit = 0): (T, Bit)
  signedAdd(other: T, overflowAction: () → T): T
  unsignedAdd(other: T, overflowAction: () → T): T
  saturatingSignedAdd(other: T): T
  saturatingUnsignedAdd(other: T): T
  wrappingSubtract(other: T): T
  subtract(other: T, carryIn: Bit = 1): (T, Bit)
  signedSubtract(other: T, overflowAction: () → T): T
  unsignedSubtract(other: T, overflowAction: () → T): T
  saturatingSignedSubtract(other: T): T
  saturatingUnsignedSubtract(other: T): T
  wrappingNegate(): T
  negate(carryIn: Bit = 1): (T, Bit)
  signedNegate(overflowAction: () → T): T
  unsignedNegate(overflowAction: () → T): T
  saturatingSignedNegate(): T
  bitNot(): T
  bitAnd(other: T): T
  bitOr(other: T): T
  bitXor(other: T): T
  bitXorNot(other: T): T
  bitNand(other: T): T
  bitNor(other: T): T
  bitAndNot(other: T): T
  bitOrNot(other: T): T
  signedMax(other: T): T
  signedMin(other: T): T
  unsignedMax(other: T): T
  unsignedMin(other: T): T
  opr =(self, other: T): Boolean
  opr ≠(self, other: T): Boolean
  signedLT(other: T): Boolean
  signedLE(other: T): Boolean
  signedGE(other: T): Boolean
  signedGT(other: T): Boolean
  unsignedLT(other: T): Boolean
  unsignedLE(other: T): Boolean
  unsignedGE(other: T): Boolean
  unsignedGT(other: T): Boolean
  signedShift(j: IndexInt): T
  signedShift(j: IndexInt, overflowAction: () → T): T
  saturatingSignedShift(j: IndexInt): T
  unsignedShift(j: IndexInt): T
  unsignedShift(j: IndexInt, overflowAction: () → T): T
  saturatingUnsignedShift(j: IndexInt): T
  bitRotate(j: IndexInt): T
  countOneBits(): IndexInt
```

```

    countLeadingZeroBits(): IndexInt
    countTrailingZeroBits(): IndexInt
    leftmostOneBit(): T
    rightmostOneBit(): T
    bitReverse(): T
    signedIndex(): IndexInt throws { IntegerOverflowException }
    unsignedIndex(): IndexInt throws { IntegerOverflowException }
    gatherBits(mask: T): T
    spreadBits(mask: T): T
    disentangleBits(mask: T): T
    intersperseBits(mask: T): T
end

```

- 40.7.1** *wrappingAdd*(other: T): T
- 40.7.2** *add*(other: T, carryIn: Bit = 0): (T, Bit)
- 40.7.3** *signedAdd*(other: T, overflowAction: () → T): T
- 40.7.4** *unsignedAdd*(other: T, overflowAction: () → T): T
- 40.7.5** *saturatingSignedAdd*(other: T): T
- 40.7.6** *saturatingUnsignedAdd*(other: T): T

[Description to be supplied.]

- 40.7.7** *wrappingSubtract*(other: T): T
- 40.7.8** *subtract*(other: T, carryIn: Bit = 1): (T, Bit)
- 40.7.9** *signedSubtract*(other: T, overflowAction: () → T): T
- 40.7.10** *unsignedSubtract*(other: T, overflowAction: () → T): T
- 40.7.11** *saturatingSignedSubtract*(other: T): T
- 40.7.12** *saturatingUnsignedSubtract*(other: T): T

[Description to be supplied.]

- 40.7.13** *wrappingNegate*(): T
- 40.7.14** *negate*(carryIn: Bit = 1): (T, Bit)
- 40.7.15** *signedNegate*(overflowAction: () → T): T
- 40.7.16** *unsignedNegate*(overflowAction: () → T): T
- 40.7.17** *saturatingSignedNegate*(): T

[Description to be supplied.]

- 40.7.18** *bitNot*(): T

[Description to be supplied.]

- 40.7.19** *bitAnd(other: T): T*
- 40.7.20** *bitOr(other: T): T*
- 40.7.21** *bitXor(other: T): T*
- 40.7.22** *bitXorNot(other: T): T*
- 40.7.23** *bitNand(other: T): T*
- 40.7.24** *bitNor(other: T): T*
- 40.7.25** *bitAndNot(other: T): T*
- 40.7.26** *bitOrNot(other: T): T*

[Description to be supplied.]

- 40.7.27** *signedMax(other: T): T*
- 40.7.28** *signedMin(other: T): T*

[Description to be supplied.]

- 40.7.29** *unsignedMax(other: T): T*
- 40.7.30** *unsignedMin(other: T): T*

[Description to be supplied.]

- 40.7.31** *opr = (self, other: T): Boolean*
- 40.7.32** *opr ≠ (self, other: T): Boolean*

[Description to be supplied.]

- 40.7.33** *signedLT(other: T): Boolean*
- 40.7.34** *signedLE(other: T): Boolean*
- 40.7.35** *signedGE(other: T): Boolean*
- 40.7.36** *signedGT(other: T): Boolean*

[Description to be supplied.]

- 40.7.37** *unsignedLT(other: T): Boolean*
- 40.7.38** *unsignedLE(other: T): Boolean*
- 40.7.39** *unsignedGE(other: T): Boolean*
- 40.7.40** *unsignedGT(other: T): Boolean*

[Description to be supplied.]

- 40.7.41** *signedShift*(*j*: IndexInt): *T*
- 40.7.42** *signedShift*(*j*: IndexInt, *overflowAction*: () → *T*): *T*
- 40.7.43** *saturatingSignedShift*(*j*: IndexInt): *T*

[Description to be supplied.]

- 40.7.44** *unsignedShift*(*j*: IndexInt): *T*
- 40.7.45** *unsignedShift*(*j*: IndexInt, *overflowAction*: () → *T*): *T*
- 40.7.46** *saturatingUnsignedShift*(*j*: IndexInt): *T*

[Description to be supplied.]

- 40.7.47** *bitRotate*(*j*: IndexInt): *T*

[Description to be supplied.]

- 40.7.48** *countOneBits*() : IndexInt

[Description to be supplied.]

- 40.7.49** *countLeadingZeroBits*() : IndexInt
- 40.7.50** *countTrailingZeroBits*() : IndexInt

[Description to be supplied.]

- 40.7.51** *leftmostOneBit*() : *T*
- 40.7.52** *rightmostOneBit*() : *T*

[Description to be supplied.]

- 40.7.53** *bitReverse*() : *T*

[Description to be supplied.]

- 40.7.54** *signedIndex*() : IndexInt throws { IntegerOverflowException }
- 40.7.55** *unsignedIndex*() : IndexInt throws { IntegerOverflowException }

[Description to be supplied.]

40.7.56 *gatherBits*(*mask*: *T*): *T*
40.7.57 *spreadBits*(*mask*: *T*): *T*

[Description to be supplied.]

40.7.58 *disentangleBits*(*mask*: *T*): *T*
40.7.59 *intersperseBits*(*mask*: *T*): *T*

[Description to be supplied.]

40.8 The Trait `Fortress.Core.BasicBinaryWordOperations`

```
trait BasicBinaryWordOperations[[T extends BasicBinaryWordOperations[[T, b], nat b]]
  extends BasicBinaryOperations[[T] where { b ≤ maxBinaryWordBitLog }
  multiplyLow(other: T): T where { b ≤ maxMultiplyBitLog }
  multiplyLow(other: T, overflowAction: () → T): T where { b ≤ maxMultiplyBitLog }
  saturatedMultiplyLow(other: T): T where { b ≤ maxMultiplyBitLog }
  multiplyHigh(other: T): T where { b ≤ maxMultiplyBitLog }
  multiplyDouble(other: T): (T, T) where { b ≤ maxMultiplyBitLog }
  signedDivide(other: T, overflowAction: () → T, zeroDivideAction: () → T): T
    where { b ≤ maxDivideBitLog }
  unsignedDivide(other: T, zeroDivideAction: () → T): T where { b ≤ maxDivideBitLog }
  signedDivRem(other: T, overflowAction: () → T, zeroDivideAction: () → T): (T, T)
    where { b ≤ maxDivideBitLog }
  unsignedDivRem(other: T, zeroDivideAction: () → T): (T, T) where { b ≤ maxDivideBitLog }
  signedRemainder(other: T, zeroDivideAction: () → T): T where { b ≤ maxDivideBitLog }
  unsignedModulo(other: T, zeroDivideAction: () → T): T where { b ≤ maxDivideBitLog }
  bitSwap(j: IndexInt): T
  getter littleEndian(): BinaryEndianWord[[b, false, false]]
  getter bigEndian(): BinaryEndianWord[[b, true, true]]
end
```

40.8.1 *multiplyLow*(*other*: *T*): *T* where { *b* ≤ *maxMultiplyBitLog* }
40.8.2 *multiplyLow*(*other*: *T*, *overflowAction*: () → *T*): *T* where { *b* ≤ *maxMultiplyBitLog* }
40.8.3 *saturatedMultiplyLow*(*other*: *T*): *T* where { *b* ≤ *maxMultiplyBitLog* }
40.8.4 *multiplyHigh*(*other*: *T*): *T* where { *b* ≤ *maxMultiplyBitLog* }
40.8.5 *multiplyDouble*(*other*: *T*): (*T*, *T*) where { *b* ≤ *maxMultiplyBitLog* }

[Description to be supplied.]

40.8.6 *signedDivide*(*other*: *T*, *overflowAction*: () → *T*, *zeroDivideAction*: () → *T*): *T*
where { *b* ≤ *maxDivideBitLog* }

40.8.7 *unsignedDivide*(*other*: *T*, *zeroDivideAction*: () → *T*): *T* where { *b* ≤ *maxDivideBitLog* }

[Description to be supplied.]

40.8.8 *signedDivRem*(*other*: *T*, *overflowAction*: () → *T*, *zeroDivideAction*: () → *T*): (*T*, *T*)
where { *b* ≤ *maxDivideBitLog* }

40.8.9 *unsignedDivRem*(*other*: *T*, *zeroDivideAction*: () → *T*): (*T*, *T*) where { *b* ≤ *maxDivideBitLog* }

[Description to be supplied.]

40.8.10 *signedRemainder*(*other*: *T*, *zeroDivideAction*: () → *T*): *T* where { *b* ≤ *maxDivideBitLog* }

40.8.11 *unsignedModulo*(*other*: *T*, *zeroDivideAction*: () → *T*): *T* where { *b* ≤ *maxDivideBitLog* }

[Description to be supplied.]

40.8.12 *bitSwap*(*j*: IndexInt): *T*

[Description to be supplied.]

40.8.13 *getter littleEndian*(): BinaryEndianWord[*b*, *false*, *false*]

40.8.14 *getter bigEndian*(): BinaryEndianWord[*b*, *true*, *true*]

[Description to be supplied.]

40.9 The Trait `Fortress.Core.BinaryLinearEndianSequence`

```

trait BinaryLinearEndianSequence[nat b, nat n, bool bigEndianSequence]
  extends { BasicBinaryOperations[BinaryLinearEndianSequence[b, n, bigEndianSequence]] }
  where { b ≤ maxBinaryWordBitLog }
  coercion [nat b', bool bigEndianBytes, bool bigEndianBits]
    (x: BinaryEndianWord[b', bigEndianBytes, bigEndianBits])
  where { bigEndianBytes = bigEndianSequence, 2b' = n · 2b }
  coercion [nat m, nat radix, nat q, nat k, nat v] (x: NaturalNumeral[m, radix, v])
  where { radix = 2q, q · m = n · 2bk }
  opr [j: IndexInt] : BinaryWord[b]
    throws { IndexOutOfBounds }
  opr [nat k][j: IntegerStatic[k] ] : BinaryWord[b] where { k < n }
  opr [nat m][r: RangeOfStaticSize[IndexInt, m]] :
    BinaryLinearEndianSequence[b, m, bigEndianSequence]
    throws { IndexOutOfBounds } where { m ≤ n }
  opr [int a, nat m, int c][r: StaticRange[a, m, c] ] :
    BinaryLinearEndianSequence[b, m, bigEndianSequence]
    where { 0 ≤ a < n, 0 ≤ a + m · c < n }
  opr [j: IndexInt] := (v: BinaryWord[b]):
    BinaryLinearEndianSequence[b, n, bigEndianSequence]
    throws { IndexOutOfBounds }
  opr [nat k][j: IntegerStatic[k] ] := (v: BinaryWord[b]):
    BinaryLinearEndianSequence[b, n, bigEndianSequence] where { k < n }
  opr [nat m][r: RangeOfStaticSize[IndexInt, m]] :=
    (v: BinaryLinearEndianSequence[b, m, bigEndianSequence]):
    BinaryLinearEndianSequence[b, n, bigEndianSequence]
    throws { IndexOutOfBounds }
  opr [int a, nat m, int c][r: StaticRange[a, m, c] ] :=
    (v: BinaryLinearEndianSequence[b, m, bigEndianSequence]):
    BinaryLinearEndianSequence[b, n, bigEndianSequence]
    where { 0 ≤ a < n, 0 ≤ a + m · c < n }
  update(j: IndexInt, v: BinaryWord[b]):
    BinaryLinearEndianSequence[b, n, bigEndianSequence]
    throws { IndexOutOfBounds }
  update[nat k](j: IntegerStatic[k] , v: BinaryWord[b]):
    BinaryLinearEndianSequence[b, n, bigEndianSequence] where { k < n }
  update[nat m](r: RangeOfStaticSize[IndexInt, m],
    v: BinaryLinearEndianSequence[b, m, bigEndianSequence]):
    BinaryLinearEndianSequence[b, n, bigEndianSequence]
    throws { IndexOutOfBounds }
  update[int a, nat m, int c](r: StaticRange[a, m, c] ,
    v: BinaryLinearEndianSequence[b, m, bigEndianSequence]):
    BinaryLinearEndianSequence[b, n, bigEndianSequence]
    where { 0 ≤ a < n, 0 ≤ a + m · c < n }
  opr || [nat m](self, other: BinaryLinearEndianSequence[b, m, bigEndianSequence]):
    BinaryLinearEndianSequence[b, n + m, bigEndianSequence]
  opr || [nat m, nat radix, nat q, nat k, nat v](self, other: NaturalNumeral[m, radix, v]):
    LinearSequence[BinaryWord[b], n + k] where { radix = 2q, q · m = k · 2b }
  opr || [nat m, nat radix, nat q, nat k, nat v](other: NaturalNumeral[m, radix, v], self):
    LinearSequence[BinaryWord[b], n + k] where { radix = 2q, q · m = k · 2b }

```

```

littleEndian() : BinaryEndianLinearEndianSequence[[b, false, false, n, bigEndianSequence]]
bigEndian() : BinaryEndianLinearEndianSequence[[b, true, true, n, bigEndianSequence]]
littleEndianBits() : BinaryEndianLinearEndianSequence[[b, bigEndianBytes, false,
                                                         n, bigEndianSequence]]
bigEndianBits() : BinaryEndianLinearEndianSequence[[b, bigEndianBytes, true,
                                                         n, bigEndianSequence]]
littleEndianSequence() : BinaryLinearEndianSequence[[b, n, false]]
bigEndianSequence() : BinaryLinearEndianSequence[[b, n, true]]
split[[nat b']]() : BinaryLinearEndianSequence[[b', n · 2b - b', bigEndianSequence]]
  where { b' ≤ b }
end

```

40.9.1 coercion [[*nat* *b'*, *bool* *bigEndianBytes*, *bool* *bigEndianBits*]]
 (*x*: BinaryEndianWord[[*b'*, *bigEndianBytes*, *bigEndianBits*]])
 where { *bigEndianBytes* = *bigEndianSequence*, 2^{*b'*} = *n* · 2^{*b*} }

[Description to be supplied.]

40.9.2 coercion [[*nat* *m*, *nat* *radix*, *nat* *q*, *nat* *k*, *nat* *v*]] (*x*: NaturalNumeral[[*m*, *radix*, *v*]])
 where { *radix* = 2^{*q*}, *q* · *m* = *n* · 2^{*b_k*} }

[Description to be supplied.]

40.9.3 opr [*j*: IndexInt] : BinaryWord[[*b*]]
 throws { IndexOutOfBoundsExcepion }
40.9.4 opr [[*nat* *k*]] [*j*: IntegerStatic[[*k*]]] : BinaryWord[[*b*]] where { *k* < *n* }

[Description to be supplied.]

40.9.5 opr [[*nat* *m*]] [*r*: RangeOfStaticSize[[IndexInt, *m*]]] :
 BinaryLinearEndianSequence[[*b*, *m*, *bigEndianSequence*]]
 throws { IndexOutOfBoundsExcepion } where { *m* ≤ *n* }
40.9.6 opr [[*int* *a*, *nat* *m*, *int* *c*]] [*r*: StaticRange[[*a*, *m*, *c*]]] :
 BinaryLinearEndianSequence[[*b*, *m*, *bigEndianSequence*]]
 where { 0 ≤ *a* < *n*, 0 ≤ *a* + *m* · *c* < *n* }

[Description to be supplied.]

40.9.7 `opr [j: IndexInt] := (v: BinaryWord[b]):`
`BinaryLinearEndianSequence[b, n, bigEndianSequence]`
`throws { IndexOutOfBounds }`

40.9.8 `opr [nat k][j: IntegerStatic[k]] := (v: BinaryWord[b]):`
`BinaryLinearEndianSequence[b, n, bigEndianSequence] where { k < n }`

[Description to be supplied.]

40.9.9 `opr [nat m][r: RangeOfStaticSize[IndexInt, m]] :=`
`(v: BinaryLinearEndianSequence[b, m, bigEndianSequence]):`
`BinaryLinearEndianSequence[b, n, bigEndianSequence]`
`throws { IndexOutOfBounds }`

40.9.10 `opr [int a, nat m, int c][r: StaticRange[a, m, c]] :=`
`(v: BinaryLinearEndianSequence[b, m, bigEndianSequence]):`
`BinaryLinearEndianSequence[b, n, bigEndianSequence]`
`where { 0 ≤ a < n, 0 ≤ a + m · c < n }`

[Description to be supplied.]

40.9.11 `update(j: IndexInt, v: BinaryWord[b]):`
`BinaryLinearEndianSequence[b, n, bigEndianSequence]`
`throws { IndexOutOfBounds }`

40.9.12 `update[nat k](j: IntegerStatic[k], v: BinaryWord[b]):`
`BinaryLinearEndianSequence[b, n, bigEndianSequence] where { k < n }`

[Description to be supplied.]

40.9.13 `update[nat m](r: RangeOfStaticSize[IndexInt, m],`
`v: BinaryLinearEndianSequence[b, m, bigEndianSequence]):`
`BinaryLinearEndianSequence[b, n, bigEndianSequence]`
`throws { IndexOutOfBounds }`

40.9.14 `update[int a, nat m, int c](r: StaticRange[a, m, c],`
`v: BinaryLinearEndianSequence[b, m, bigEndianSequence]):`
`BinaryLinearEndianSequence[b, n, bigEndianSequence]`
`where { 0 ≤ a < n, 0 ≤ a + m · c < n }`

[Description to be supplied.]

40.9.15 `opr || [nat m](self, other: BinaryLinearEndianSequence[b, m, bigEndianSequence]):`
`BinaryLinearEndianSequence[b, n + m, bigEndianSequence]`

[Description to be supplied.]

40.10 The Trait `Fortress.Core.BinaryEndianLinearEndianSequence`

```

trait BinaryEndianLinearEndianSequence[[nat b, bool bigEndianBytes, bool bigEndianBits,
                                       nat n, bool bigEndianSequence]]
  extends { BasicBinaryOperations[
    BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
                                       n, bigEndianSequence]] }

  where { b ≤ maxBinaryWordBitLog }
  coercion [[nat b', bool bigEndianBytes, bool bigEndianBits]
    (x: BinaryEndianWord[[b', bigEndianBytes, bigEndianBits]])
    where { bigEndianBytes = bigEndianSequence, 2b' = n · 2b }
  coercion [[nat m, nat radix, nat q, nat k, nat v](x: NaturalNumeral[[m, radix, v]])
    where { radix = 2q, q · m = n · 2bk }
  opr [j: IndexInt] : BinaryEndianWord[[b, bigEndianBytes, bigEndianBits]]
    throws { IndexOutOfBoundsException }
  opr [[nat k][j: IntegerStatic[[k]]] : BinaryWord[[b, bigEndianBytes, bigEndianBits]]
    where { k < n }
  opr [[nat m][r: RangeOfStaticSize[[IndexInt, m]]] :
    BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
                                       m, bigEndianSequence]]
    throws { IndexOutOfBoundsException } where { m ≤ n }
  opr [[int a, nat m, int c][r: StaticRange[[a, m, c]]] :
    BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
                                       m, bigEndianSequence]]
    where { 0 ≤ a < n, 0 ≤ a + m · c < n }
  opr [j: IndexInt] := (v: BinaryEndianWord[[b, bigEndianBytes, bigEndianBits]]):
    BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
                                       n, bigEndianSequence]]
    throws { IndexOutOfBoundsException }
  opr [[nat k][j: IntegerStatic[[k]]] :=
    (v: BinaryEndianWord[[b, bigEndianBytes, bigEndianBits]]):
    BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
                                       n, bigEndianSequence]]
    where { k < n }
  opr [[nat m][r: RangeOfStaticSize[[IndexInt, m]]] :=
    (v: BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
                                       m, bigEndianSequence]]):
    BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
                                       n, bigEndianSequence]]
    throws { IndexOutOfBoundsException }
  opr [[int a, nat m, int c][r: StaticRange[[a, m, c]]] :=
    (v: BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
                                       k, bigEndianSequence]]):
    BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
                                       n, bigEndianSequence]]
    where { 0 ≤ a < n, 0 ≤ a + m · c < n }
  update(j: IndexInt, v: BinaryEndianWord[[b, bigEndianBytes, bigEndianBits]]):
    BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
                                       n, bigEndianSequence]]
    throws { IndexOutOfBoundsException }
  update[[nat k](j: IntegerStatic[[k]],

```

```

        v: BinaryEndianWord[[b, bigEndianBytes, bigEndianBits]]):
    BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
        n, bigEndianSequence]]

    where { k < n }
update[[nat m]](r: RangeOfStaticSize[[IndexInt, m]],
    v: BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
        m, bigEndianSequence]]):

    BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
        n, bigEndianSequence]]

    throws { IndexOutOfBoundsException }
update[[int a, nat m, int c]]
    (r: StaticRange[[a, m, c]],
    v: BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
        m, bigEndianSequence]]):

    BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
        n, bigEndianSequence]]

    where { 0 ≤ a < n, 0 ≤ a + m · c < n }
opr || [[nat m]](self, other: BinaryEndianLinearEndianSequence[[
        b, bigEndianBytes, bigEndianBits,
        m, bigEndianSequence]]):

    BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
        n + m, bigEndianSequence]]

opr || [[nat m, nat radix, nat q, nat k, nat v]](self, other: NaturalNumeral[[m, radix, v]]):
    LinearSequence[[T, n + k]]
    where { radix = 2q, q · m = k · 2b }
opr || [[nat m, nat radix, nat q, nat k, nat v]](other: NaturalNumeral[[m, radix, v]], self):
    LinearSequence[[T, n + k]]
    where { radix = 2q, q · m = k · 2b }
littleEndian(): BinaryEndianLinearEndianSequence[[b, false, false, n, bigEndianSequence]]
bigEndian(): BinaryEndianLinearEndianSequence[[b, true, true, n, bigEndianSequence]]
littleEndianBits(): BinaryEndianLinearEndianSequence[[b, bigEndianBytes, false,
    n, bigEndianSequence]]
bigEndianBits(): BinaryEndianLinearEndianSequence[[b, bigEndianBytes, true,
    n, bigEndianSequence]]
littleEndianSequence(): BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
    n, false]]
bigEndianSequence(): BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
    n, true]]

split[[nat b']]():
    BinaryEndianLinearEndianSequence[[b', bigEndianBytes, bigEndianBits,
        n · 2b-b', bigEndianSequence]]

    where { b' ≤ b }
end

```

40.10.1 coercion [[nat b', bool bigEndianBytes, bool bigEndianBits]]
(x: BinaryEndianWord[[b', bigEndianBytes, bigEndianBits]])
where { bigEndianBytes = bigEndianSequence, 2^{b'} = n · 2^b }

[Description to be supplied.]

40.10.2 coercion $\llbracket \text{nat } m, \text{nat } radix, \text{nat } q, \text{nat } k, \text{nat } v \rrbracket (x: \text{NaturalNumeral} \llbracket m, radix, v \rrbracket)$
 where $\{ radix = 2^q, q \cdot m = n \cdot 2^{bk} \}$

[Description to be supplied.]

40.10.3 opr $\llbracket j: \text{IndexInt} \rrbracket : \text{BinaryEndianWord} \llbracket b, bigEndianBytes, bigEndianBits \rrbracket$
 throws $\{ \text{IndexOutOfBounds} \}$
40.10.4 opr $\llbracket \text{nat } k \rrbracket \llbracket j: \text{IntegerStatic} \llbracket k \rrbracket \rrbracket : \text{BinaryWord} \llbracket b, bigEndianBytes, bigEndianBits \rrbracket$
 where $\{ k < n \}$

[Description to be supplied.]

40.10.5 opr $\llbracket \text{nat } m \rrbracket \llbracket r: \text{RangeOfStaticSize} \llbracket \text{IndexInt}, m \rrbracket \rrbracket : \text{BinaryEndianLinearEndianSequence} \llbracket b, bigEndianBytes, bigEndianBits, m, bigEndianSequence \rrbracket$
 throws $\{ \text{IndexOutOfBounds} \}$ where $\{ m \leq n \}$
40.10.6 opr $\llbracket \text{int } a, \text{nat } m, \text{int } c \rrbracket \llbracket r: \text{StaticRange} \llbracket a, m, c \rrbracket \rrbracket : \text{BinaryEndianLinearEndianSequence} \llbracket b, bigEndianBytes, bigEndianBits, m, bigEndianSequence \rrbracket$
 where $\{ 0 \leq a < n, 0 \leq a + m \cdot c < n \}$

[Description to be supplied.]

40.10.7 opr $\llbracket j: \text{IndexInt} \rrbracket := (v: \text{BinaryEndianWord} \llbracket b, bigEndianBytes, bigEndianBits \rrbracket) : \text{BinaryEndianLinearEndianSequence} \llbracket b, bigEndianBytes, bigEndianBits, n, bigEndianSequence \rrbracket$
 throws $\{ \text{IndexOutOfBounds} \}$
40.10.8 opr $\llbracket \text{nat } k \rrbracket \llbracket j: \text{IntegerStatic} \llbracket k \rrbracket \rrbracket := (v: \text{BinaryEndianWord} \llbracket b, bigEndianBytes, bigEndianBits \rrbracket) : \text{BinaryEndianLinearEndianSequence} \llbracket b, bigEndianBytes, bigEndianBits, n, bigEndianSequence \rrbracket$
 where $\{ k < n \}$

[Description to be supplied.]

- 40.10.9** `opr` $[[\text{nat } m][r: \text{RangeOfStaticSize}[[\text{IndexInt}, m]] :=$
 $(v: \text{BinaryEndianLinearEndianSequence}[[b, \text{bigEndianBytes}, \text{bigEndianBits},$
 $m, \text{bigEndianSequence}]]):$
 $\text{BinaryEndianLinearEndianSequence}[[b, \text{bigEndianBytes}, \text{bigEndianBits},$
 $n, \text{bigEndianSequence}]$
`throws` { `IndexOutOfBoundsException` }
- 40.10.10** `opr` $[[\text{int } a, \text{nat } m, \text{int } c][r: \text{StaticRange}[[a, m, c]] :=$
 $(v: \text{BinaryEndianLinearEndianSequence}[[b, \text{bigEndianBytes}, \text{bigEndianBits},$
 $k, \text{bigEndianSequence}]]):$
 $\text{BinaryEndianLinearEndianSequence}[[b, \text{bigEndianBytes}, \text{bigEndianBits},$
 $n, \text{bigEndianSequence}]$
`where` { $0 \leq a < n, 0 \leq a + m \cdot c < n$ }

[Description to be supplied.]

- 40.10.11** `update` $(j: \text{IndexInt}, v: \text{BinaryEndianWord}[[b, \text{bigEndianBytes}, \text{bigEndianBits}]]):$
 $\text{BinaryEndianLinearEndianSequence}[[b, \text{bigEndianBytes}, \text{bigEndianBits},$
 $n, \text{bigEndianSequence}]$
`throws` { `IndexOutOfBoundsException` }
- 40.10.12** `update` $[[\text{nat } k](j: \text{IntegerStatic}[[k],$
 $v: \text{BinaryEndianWord}[[b, \text{bigEndianBytes}, \text{bigEndianBits}]]):$
 $\text{BinaryEndianLinearEndianSequence}[[b, \text{bigEndianBytes}, \text{bigEndianBits},$
 $n, \text{bigEndianSequence}]$
`where` { $k < n$ }

[Description to be supplied.]

- 40.10.13** `update` $[[\text{nat } m](r: \text{RangeOfStaticSize}[[\text{IndexInt}, m],$
 $v: \text{BinaryEndianLinearEndianSequence}[[b, \text{bigEndianBytes}, \text{bigEndianBits},$
 $m, \text{bigEndianSequence}]]):$
 $\text{BinaryEndianLinearEndianSequence}[[b, \text{bigEndianBytes}, \text{bigEndianBits},$
 $n, \text{bigEndianSequence}]$
`throws` { `IndexOutOfBoundsException` }
- 40.10.14** `update` $[[\text{int } a, \text{nat } m, \text{int } c]$
 $(r: \text{StaticRange}[[a, m, c],$
 $v: \text{BinaryEndianLinearEndianSequence}[[b, \text{bigEndianBytes}, \text{bigEndianBits},$
 $m, \text{bigEndianSequence}]]):$
 $\text{BinaryEndianLinearEndianSequence}[[b, \text{bigEndianBytes}, \text{bigEndianBits},$
 $n, \text{bigEndianSequence}]$
`where` { $0 \leq a < n, 0 \leq a + m \cdot c < n$ }

[Description to be supplied.]

40.10.15 `opr` || `[[nat m]](self, other: BinaryEndianLinearEndianSequence[[
b, bigEndianBytes, bigEndianBits,
m, bigEndianSequence]]):
BinaryEndianLinearEndianSequence[[b, bigEndianBytes, bigEndianBits,
n + m, bigEndianSequence]]`

[Description to be supplied.]

40.10.16 `opr` || `[[nat m, nat radix, nat q, nat k, nat v]](self, other: NaturalNumeral[[m, radix, v]]):
LinearSequence[[T, n + k]]
where { radix = 2^q , $q \cdot m = k \cdot 2^b$ }`

[Description to be supplied.]

40.10.17 `opr` || `[[nat m, nat radix, nat q, nat k, nat v]](other: NaturalNumeral[[m, radix, v]], self):
LinearSequence[[T, n + k]]
where { radix = 2^q , $q \cdot m = k \cdot 2^b$ }`

[Description to be supplied.]

40.10.18 `littleEndian()`: BinaryEndianLinearEndianSequence[[*b*, *false*, *false*, *n*, *bigEndianSequence*]]
40.10.19 `bigEndian()`: BinaryEndianLinearEndianSequence[[*b*, *true*, *true*, *n*, *bigEndianSequence*]]

[Description to be supplied.]

40.10.20 `littleEndianBits()`: BinaryEndianLinearEndianSequence[[*b*, *bigEndianBytes*, *false*,
n, *bigEndianSequence*]]
40.10.21 `bigEndianBits()`: BinaryEndianLinearEndianSequence[[*b*, *bigEndianBytes*, *true*,
n, *bigEndianSequence*]]

[Description to be supplied.]

40.10.22 `littleEndianSequence()`: BinaryEndianLinearEndianSequence[[*b*, *bigEndianBytes*, *bigEndianBits*,
n, *false*]]
40.10.23 `bigEndianSequence()`: BinaryEndianLinearEndianSequence[[*b*, *bigEndianBytes*, *bigEndianBits*,
n, *true*]]

[Description to be supplied.]

40.10.24 *split*[[nat b']]():
 BinaryEndianLinearEndianSequence[[b' , *bigEndianBytes*, *bigEndianBits*,
 $n \cdot 2^{b-b'}$, *bigEndianSequence*]]
 where { $b' \leq b$ }

[Description to be supplied.]

40.11 The Trait Fortress.Core.BinaryHeapEndianSequence

```

trait BinaryHeapEndianSequence[[nat  $b$ , bool bigEndianSequence]]
  extends { BinaryHeapSequence[[ $b$ ],
    BasicBinaryHeapSubsequenceOperations[[
      BinaryHeapEndianSequence[[ $b$ , bigEndianSequence]],
      bigEndianSequence]] }
end

```

40.12 The Trait Fortress.Core.BinaryEndianHeapEndianSequence

```

trait BinaryEndianHeapEndianSequence[[nat  $b$ , bool bigEndianBytes, bool bigEndianBits,
  bool bigEndianSequence]]
  extends { HeapSequence[[BinaryEndianWord[[ $b$ , bigEndianBytes, bigEndianBits]]],
    BasicBinaryHeapSubsequenceOperations[[
      BinaryHeapEndianSequence[[ $b$ , bigEndianSequence]],
      bigEndianSequence]] }
end

```

40.13 The Trait `Fortress.Core.BasicBinaryHeapSubsequenceOperations`

```
trait BasicBinaryHeapSubsequenceOperations[[
  T extends BasicBinaryHeapSubsequenceOperations[[T, bigEndianSequence]],
  bool bigEndianSequence]]
copy(selfStart: IndexInt, source: T, sourceStart: IndexInt, length: IndexInt): ()
  throws { IndexOutOfBounds }
wrappingAdd(selfStart: IndexInt, source: T, sourceStart: IndexInt, length: IndexInt): ()
  throws { IndexOutOfBounds }
add(selfStart: IndexInt, source: T, sourceStart: IndexInt,
    length: IndexInt, carryIn: Bit = 0): Bit
  throws { IndexOutOfBounds }
signedAdd(selfStart: IndexInt, source: T, sourceStart: IndexInt,
    length: IndexInt, overflowAction: () → ()): ()
  throws { IndexOutOfBounds }
unsignedAdd(selfStart: IndexInt, source: T, sourceStart: IndexInt,
    length: IndexInt, overflowAction: () → ()): ()
  throws { IndexOutOfBounds }
saturatingSignedAdd(selfStart: IndexInt, source: T, sourceStart: IndexInt, length: IndexInt): ()
  throws { IndexOutOfBounds }
saturatingUnsignedAdd(selfStart: IndexInt, source: T, sourceStart: IndexInt, length: IndexInt): ()
  throws { IndexOutOfBounds }
wrappingSubtract(selfStart: IndexInt, source: T, sourceStart: IndexInt, length: IndexInt): ()
  throws { IndexOutOfBounds }
subtract(selfStart: IndexInt, source: T, sourceStart: IndexInt,
    length: IndexInt, carryIn: Bit = 1): Bit
  throws { IndexOutOfBounds }
signedSubtract(selfStart: IndexInt, source: T, sourceStart: IndexInt,
    length: IndexInt, overflowAction: () → ()): ()
  throws { IndexOutOfBounds }
unsignedSubtract(selfStart: IndexInt, source: T, sourceStart: IndexInt,
    length: IndexInt, overflowAction: () → ()): ()
  throws { IndexOutOfBounds }
saturatingSignedSubtract(selfStart: IndexInt, source: T, sourceStart: IndexInt,
    length: IndexInt): ()
  throws { IndexOutOfBounds }
saturatingUnsignedSubtract(selfStart: IndexInt, source: T, sourceStart: IndexInt,
    length: IndexInt): ()
  throws { IndexOutOfBounds }
wrappingNegate(selfStart: IndexInt, length: IndexInt): ()
  throws { IndexOutOfBounds }
negate(selfStart: IndexInt, length: IndexInt, carryIn: Bit = 1): Bit
  throws { IndexOutOfBounds }
signedNegate(selfStart: IndexInt, length: IndexInt, overflowAction: () → ()): ()
  throws { IndexOutOfBounds }
unsignedNegate(selfStart: IndexInt, length: IndexInt, overflowAction: () → ()): ()
  throws { IndexOutOfBounds }
saturatingSignedNegate(selfStart: IndexInt, length: IndexInt): ()
  throws { IndexOutOfBounds }
bitNot(selfStart: IndexInt, length: IndexInt): ()
  throws { IndexOutOfBounds }
```



```

unsignedShift(selfStart: IndexInt, length: IndexInt, j: IndexInt, overflowAction: () → ()): ()
    throws { IndexOutOfBounds }
saturatingUnsignedShift(selfStart: IndexInt, length: IndexInt, j: IndexInt): ()
    throws { IndexOutOfBounds }
bitRotate(selfStart: IndexInt, length: IndexInt, j: IndexInt): ()
    throws { IndexOutOfBounds }
countOneBits(selfStart: IndexInt, length: IndexInt): IndexInt
    throws { IndexOutOfBounds }
countLeadingZeroBits(selfStart: IndexInt, length: IndexInt): IndexInt
    throws { IndexOutOfBounds }
countTrailingZeroBits(selfStart: IndexInt, length: IndexInt): IndexInt
    throws { IndexOutOfBounds }
leftmostOneBit(selfStart: IndexInt, length: IndexInt): ()
    throws { IndexOutOfBounds }
rightmostOneBit(selfStart: IndexInt, length: IndexInt): ()
    throws { IndexOutOfBounds }
bitReverse(selfStart: IndexInt, length: IndexInt): ()
    throws { IndexOutOfBounds }
gatherBits(selfStart: IndexInt, mask: T, maskStart: IndexInt, length: IndexInt): ()
    throws { IndexOutOfBounds }
spreadBits(selfStart: IndexInt, mask: T, maskStart: IndexInt, length: IndexInt): ()
    throws { IndexOutOfBounds }
clearAllBits(selfStart: IndexInt, length: IndexInt): ()
    throws { IndexOutOfBounds }
setAllBits(selfStart: IndexInt, length: IndexInt): ()
    throws { IndexOutOfBounds }
signedIndex(): IndexInt
    throws { IntegerOverflow }
signedIndex(selfStart: IndexInt, length: IndexInt): IndexInt
    throws { IndexOutOfBounds, IntegerOverflow }
unsignedIndex(): IndexInt
    throws { IntegerOverflow }
unsignedIndex(selfStart: IndexInt, length: IndexInt): IndexInt
    throws { IndexOutOfBounds, IntegerOverflow }
end

```

40.13.1 *copy*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
 throws { IndexOutOfBounds }

[Description to be supplied.]

- 40.13.2** *wrappingAdd*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }
- 40.13.3** *add*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt,
length: IndexInt, *carryIn*: Bit = 0): Bit
throws { IndexOutOfBoundsException }
- 40.13.4** *signedAdd*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt,
length: IndexInt, *overflowAction*: () → ()): ()
throws { IndexOutOfBoundsException }
- 40.13.5** *unsignedAdd*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt,
length: IndexInt, *overflowAction*: () → ()): ()
throws { IndexOutOfBoundsException }
- 40.13.6** *saturatingSignedAdd*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }
- 40.13.7** *saturatingUnsignedAdd*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }

[Description to be supplied.]

- 40.13.8** *wrappingSubtract*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }
- 40.13.9** *subtract*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt,
length: IndexInt, *carryIn*: Bit = 1): Bit
throws { IndexOutOfBoundsException }
- 40.13.10** *signedSubtract*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt,
length: IndexInt, *overflowAction*: () → ()): ()
throws { IndexOutOfBoundsException }
- 40.13.11** *unsignedSubtract*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt,
length: IndexInt, *overflowAction*: () → ()): ()
throws { IndexOutOfBoundsException }
- 40.13.12** *saturatingSignedSubtract*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt,
length: IndexInt): ()
throws { IndexOutOfBoundsException }
- 40.13.13** *saturatingUnsignedSubtract*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt,
length: IndexInt): ()
throws { IndexOutOfBoundsException }

[Description to be supplied.]

- 40.13.14** *wrappingNegate*(*selfStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }
- 40.13.15** *negate*(*selfStart*: IndexInt, *length*: IndexInt, *carryIn*: Bit = 1): Bit
throws { IndexOutOfBoundsException }
- 40.13.16** *signedNegate*(*selfStart*: IndexInt, *length*: IndexInt, *overflowAction*: () → ()): ()
throws { IndexOutOfBoundsException }
- 40.13.17** *unsignedNegate*(*selfStart*: IndexInt, *length*: IndexInt, *overflowAction*: () → ()): ()
throws { IndexOutOfBoundsException }
- 40.13.18** *saturatingSignedNegate*(*selfStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }

[Description to be supplied.]

- 40.13.19** *bitNot*(*selfStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }

[Description to be supplied.]

- 40.13.20** *bitAnd*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }
- 40.13.21** *bitOr*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }
- 40.13.22** *bitXor*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }
- 40.13.23** *bitXorNot*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }
- 40.13.24** *bitNand*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }
- 40.13.25** *bitNor*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }
- 40.13.26** *bitAndNot*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }
- 40.13.27** *bitOrNot*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }

[Description to be supplied.]

- 40.13.28** *signedMax*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }
- 40.13.29** *signedMin*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }

[Description to be supplied.]

- 40.13.30** *unsignedMax*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBounds }
40.13.31 *unsignedMin*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBounds }

[Description to be supplied.]

- 40.13.32** *equal*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
throws { IndexOutOfBounds }
40.13.33 *unequal*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
throws { IndexOutOfBounds }

[Description to be supplied.]

- 40.13.34** *signedLT*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
throws { IndexOutOfBounds }
40.13.35 *signedLE*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
throws { IndexOutOfBounds }
40.13.36 *signedGE*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
throws { IndexOutOfBounds }
40.13.37 *signedGT*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
throws { IndexOutOfBounds }

[Description to be supplied.]

- 40.13.38** *unsignedLT*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
throws { IndexOutOfBounds }
40.13.39 *unsignedLE*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
throws { IndexOutOfBounds }
40.13.40 *unsignedGE*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
throws { IndexOutOfBounds }
40.13.41 *unsignedGT*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
throws { IndexOutOfBounds }

[Description to be supplied.]

- 40.13.42** *signedShift*(*selfStart*: IndexInt, *length*: IndexInt, *j*: IndexInt): ()
throws { IndexOutOfBounds }
40.13.43 *signedShift*(*selfStart*: IndexInt, *length*: IndexInt, *j*: IndexInt, *overflowAction*: () → ()): ()
throws { IndexOutOfBounds }
40.13.44 *saturatingSignedShift*(*selfStart*: IndexInt, *length*: IndexInt, *j*: IndexInt): ()
throws { IndexOutOfBounds }

[Description to be supplied.]

- 40.13.45** *unsignedShift*(*selfStart*: IndexInt, *length*: IndexInt, *j*: IndexInt): ()
throws { IndexOutOfBoundsException }
- 40.13.46** *unsignedShift*(*selfStart*: IndexInt, *length*: IndexInt, *j*: IndexInt, *overflowAction*: () → ()): ()
throws { IndexOutOfBoundsException }
- 40.13.47** *saturatingUnsignedShift*(*selfStart*: IndexInt, *length*: IndexInt, *j*: IndexInt): ()
throws { IndexOutOfBoundsException }

[Description to be supplied.]

- 40.13.48** *bitRotate*(*selfStart*: IndexInt, *length*: IndexInt, *j*: IndexInt): ()
throws { IndexOutOfBoundsException }

[Description to be supplied.]

- 40.13.49** *countOneBits*(*selfStart*: IndexInt, *length*: IndexInt): IndexInt
throws { IndexOutOfBoundsException }

[Description to be supplied.]

- 40.13.50** *countLeadingZeroBits*(*selfStart*: IndexInt, *length*: IndexInt): IndexInt
throws { IndexOutOfBoundsException }
- 40.13.51** *countTrailingZeroBits*(*selfStart*: IndexInt, *length*: IndexInt): IndexInt
throws { IndexOutOfBoundsException }

[Description to be supplied.]

- 40.13.52** *leftmostOneBit*(*selfStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }
- 40.13.53** *rightmostOneBit*(*selfStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }

[Description to be supplied.]

- 40.13.54** *bitReverse*(*selfStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }

[Description to be supplied.]

- 40.13.55** *gatherBits*(*selfStart*: IndexInt, *mask*: T, *maskStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }
- 40.13.56** *spreadBits*(*selfStart*: IndexInt, *mask*: T, *maskStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }

[Description to be supplied.]

- 40.13.57** *clearAllBits*(*selfStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }
- 40.13.58** *setAllBits*(*selfStart*: IndexInt, *length*: IndexInt): ()
throws { IndexOutOfBoundsException }

[Description to be supplied.]

- 40.13.59** *signedIndex*(): IndexInt
throws { IntegerOverflowException }
- 40.13.60** *signedIndex*(*selfStart*: IndexInt, *length*: IndexInt): IndexInt
throws { IndexOutOfBoundsException, IntegerOverflowException }
- 40.13.61** *unsignedIndex*(): IndexInt
throws { IntegerOverflowException }
- 40.13.62** *unsignedIndex*(*selfStart*: IndexInt, *length*: IndexInt): IndexInt
throws { IndexOutOfBoundsException, IntegerOverflowException }

[Description to be supplied.]

Part VI

Appendices

Appendix A

Fortress Calculi

A.1 Basic Core Fortress

In this section, we define a basic core calculus for Fortress. We call this calculus *Basic Core Fortress*. Following the precedent set by prior core calculi such as Featherweight Generic Java [14], we have abided by the restriction that all valid Basic Core Fortress programs are valid Fortress programs.

A.1.1 Syntax

A syntax for Basic Core Fortress is provided in Figure A.1. We use the following notational conventions:

- For brevity, we omit separators such as `,` and `;` from Basic Core Fortress.
- $\vec{\tau}$ is a shorthand for a (possibly empty) sequence τ_1, \dots, τ_n .
- Similarly, we abbreviate a sequence of relations $\alpha_1 \text{ extends } N_1, \dots, \alpha_n \text{ extends } N_n$ to $\overline{\alpha \text{ extends } \vec{N}}$
- We use τ_i to denote the i th element of $\vec{\tau}$.
- For simplicity, we assume that every name (type variables, field names, and parameters) is different and every trait/object declaration declares unique name.
- We prohibit cycles in type hierarchies.

The syntax of Basic Core Fortress allows only a small subset of the Fortress language to be formalized. Basic Core Fortress includes trait and object definitions, method and field invocations, and `self` expressions. The types of Basic Core Fortress include type variables, instantiated traits, instantiated objects, and the distinguished trait `Object`. Note that we syntactically prohibit extending objects. Among other features, Basic Core Fortress does *not* include top-level variable and function definitions, overloading, `excludes` clauses, `bounds` clauses, `where` clauses, object expressions, and function expressions. Basic Core Fortress will be extended to formalize a larger set of Fortress programs in the future.

A.1.2 Dynamic Semantics

A dynamic semantics for Basic Core Fortress is provided in Figure A.2. This semantics has been mechanized via the PLT Redex tool [17]. It therefore follows the style of explicit evaluation contexts and redexes. The Basic Core

α, β		type variables
f		method name
x		field name
T		trait name
O		object name
τ, τ', τ''	$::= \alpha$	type
	σ	
σ	$::= N$	type that is not a type variable
	$O[\vec{\tau}]$	
N, M, L	$::= T[\vec{\tau}]$	type that can be a type bound
	Object	
e	$::= x$	expression
	self	
	$O[\vec{\tau}](\vec{e})$	
	$e.x$	
	$e.f[\vec{\tau}](\vec{e})$	
fd	$::= f[\overrightarrow{\alpha \text{ extends } N}](\overrightarrow{x:\vec{\tau}}):\tau = e$	method definition
td	$::= \text{trait } T[\overrightarrow{\alpha \text{ extends } N}] \text{ extends } \{\overrightarrow{N}\} \overrightarrow{fd} \text{ end}$	trait definition
od	$::= \text{object } O[\overrightarrow{\alpha \text{ extends } N}](\overrightarrow{x:\vec{\tau}}) \text{ extends } \{\overrightarrow{N}\} \overrightarrow{fd} \text{ end}$	object definition
d	$::= td$	definition
	od	
p	$::= \overrightarrow{d} e$	program

Figure A.1: Syntax of Basic Core Fortress

Fortress dynamic semantics consists of two evaluations rules: one for field access and another for method invocation. For simplicity, we use ‘ $_$ ’ to denote some parts of the syntax that do not have key roles in a rule. We assume that $_$ does not expand across definition boundaries unless the entire definition is included in it.

A.1.3 Static Semantics

A static semantics for Basic Core Fortress is provided in Figures A.3, A.4, and A.5. The Basic Core Fortress static semantics is based on the static semantics of Featherweight Generic Java (FGJ) [14]. The major difference is the division of classes into traits and objects. Both trait and object definitions include method definitions but only object definitions include field definitions. With traits, Basic Core Fortress supports multiple inheritance. However, due to the similarity of traits and objects, many of the rules in the Basic Core Fortress dynamic and static semantics combine the two cases. Note that Basic Core Fortress allows parametric polymorphism, subtype polymorphism, and overriding in much the same way that FGJ does.

We proved the type soundness of Basic Core Fortress using the standard technique of proving a progress theorem and a subject reduction theorem.

Values, evaluation contexts, redexes, and trait and object names

v	$::= O[\vec{\tau}](\vec{v})$	value
E	$::= \square$	evaluation context
	$ O[\vec{\tau}](\vec{e} E \vec{e})$	
	$ E.x$	
	$ E.f[\vec{\tau}](\vec{e})$	
	$ e.f[\vec{\tau}](\vec{e} E \vec{e})$	
R	$::= v.x$	redex
	$ v.f[\vec{\tau}](\vec{v})$	
C	$::= T$	trait name
	$ O$	object name

Evaluation rules: $\boxed{p \vdash E[R] \longrightarrow E[e]}$

$$\text{[R-FIELD]} \quad \frac{\text{object } O _ (\vec{x}:\vec{_}) _ \text{end} \in p}{p \vdash E[O[\vec{\tau}](\vec{v}).x_i] \longrightarrow E[v_i]}$$

$$\text{[R-METHOD]} \quad \frac{\text{object } O _ (\vec{x}:\vec{_}) _ \text{end} \in p \quad \text{mbody}_p(f[\vec{\tau}'], O[\vec{\tau}]) = \{(\vec{x}') \rightarrow e\}}{p \vdash E[O[\vec{\tau}](\vec{v}).f[\vec{\tau}'](\vec{v}')] \longrightarrow E[\vec{v}'/\vec{x}][O[\vec{\tau}](\vec{v}')/\text{self}][\vec{v}'/\vec{x}']e]}$$

Method body lookup: $\boxed{\text{mbody}_p(f[\vec{\tau}], \tau) = \{(\vec{x}') \rightarrow e\}}$

$$\text{[MB-SELF]} \quad \frac{_ C[\vec{\alpha} \text{ extends } \vec{_}] _ \vec{fd} _ \in p \quad f[\vec{\alpha}' \text{ extends } \vec{_}](\vec{x}':\vec{_}) _ = e \in \{\vec{fd}\}}{\text{mbody}_p(f[\vec{\tau}'], C[\vec{\tau}]) = \{[\vec{\tau}'/\vec{\alpha}'][\vec{\tau}/\vec{\alpha}](\vec{x}') \rightarrow e\}}$$

$$\text{[MB-SUPER]} \quad \frac{_ C[\vec{\alpha} \text{ extends } \vec{_}] _ \text{extends } \{\vec{N}\} _ \vec{fd} _ \in p \quad f \notin \{\overline{Fname}(fd)\}}{\text{mbody}_p(f[\vec{\tau}'], C[\vec{\tau}]) = \bigcup_{N_i \in \{\vec{N}\}} \text{mbody}_p(f[\vec{\tau}'], [\vec{\tau}/\vec{\alpha}]N_i)}$$

$$\text{[MB-OBJ]} \quad \text{mbody}_p(f[\vec{\tau}], \text{Object}) = \emptyset$$

Function/method name lookup: $\boxed{Fname(fd) = f}$

$$Fname(f[\vec{\alpha} \text{ extends } \vec{N}](\vec{x}:\vec{\tau}) : \tau = e) = f$$

Figure A.2: Dynamic Semantics of Basic Core Fortress

Environments

$$\begin{array}{ll} \Delta ::= \overline{\alpha <: N} & \text{bound environment} \\ \Gamma ::= \overline{x : \vec{\tau}} & \text{type environment} \end{array}$$

Program typing: $\boxed{\vdash p : \tau}$

$$[\text{T-PROGRAM}] \quad \frac{p = \overrightarrow{d} \ e \quad p \vdash \overrightarrow{d} \text{ ok} \quad p; \emptyset; \emptyset \vdash e : \tau}{\vdash p : \tau}$$

Definition typing: $\boxed{p \vdash d \text{ ok}}$

$$[\text{T-TRAITDEF}] \quad \frac{\Delta = \overline{\alpha <: N} \quad p; \Delta \vdash \overrightarrow{N} \text{ ok} \quad p; \Delta \vdash \overrightarrow{M} \text{ ok} \quad p; \Delta; \text{self} : T[\overline{\alpha}]; T \vdash \overrightarrow{fd} \text{ ok} \quad p \vdash \text{oneOwner}(T)}{p \vdash \text{trait } T[\overline{\alpha \text{ extends } N}] \text{ extends } \{\overrightarrow{M}\} \overrightarrow{fd} \text{ end ok}}$$

$$[\text{T-OBJECTDEF}] \quad \frac{\Delta = \overline{\alpha <: N} \quad p; \Delta \vdash \overrightarrow{N} \text{ ok} \quad p; \Delta \vdash \overrightarrow{\tau} \text{ ok} \quad p; \Delta \vdash \overrightarrow{M} \text{ ok} \quad p; \Delta; \text{self} : O[\overline{\alpha}] \ \overline{x : \vec{\tau}}; O \vdash \overrightarrow{fd} \text{ ok} \quad p \vdash \text{oneOwner}(O)}{p \vdash \text{object } O[\overline{\alpha \text{ extends } N}](\overline{x : \vec{\tau}}) \text{ extends } \{\overrightarrow{M}\} \overrightarrow{fd} \text{ end ok}}$$

Method typing: $\boxed{p; \Delta; \Gamma; C \vdash fd \text{ ok}}$

$$[\text{T-METHODDEF}] \quad \frac{\begin{array}{l} - C _ \text{ extends } \{\overrightarrow{M}\} _ \in p \quad p; \Delta \vdash \text{override}(f, \{\overrightarrow{M}\}, [\overline{\alpha \text{ extends } N}]) \ \overrightarrow{\tau} \rightarrow \tau_0 \\ \Delta' = \Delta \ \overline{\alpha <: N} \quad p; \Delta' \vdash \overrightarrow{N} \text{ ok} \quad p; \Delta' \vdash \overrightarrow{\tau} \text{ ok} \quad p; \Delta' \vdash \tau_0 \text{ ok} \\ p; \Delta'; \Gamma \ \overline{x : \vec{\tau}} \vdash e : \tau' \quad p; \Delta' \vdash \tau' <: \tau_0 \end{array}}{p; \Delta; \Gamma; C \vdash f[\overline{\alpha \text{ extends } N}](\overline{x : \vec{\tau}}) : \tau_0 = e \text{ ok}}$$

Method overriding: $\boxed{p; \Delta \vdash \text{override}(f, \{\overrightarrow{N}\}, [\overline{\alpha \text{ extends } N}]) \ \overrightarrow{\tau} \rightarrow \tau}$

$$[\text{OVERRIDE}] \quad \frac{\bigcup_{L_i \in \{\overrightarrow{L}\}} \text{mtype}_p(f, L_i) = \{[\overline{\beta \text{ extends } M}] \ \overrightarrow{\tau}' \rightarrow \tau'_0\} \quad \overrightarrow{N} = [\overline{\alpha / \beta}] \overrightarrow{M} \quad \overrightarrow{\tau} = [\overline{\alpha / \beta}] \overrightarrow{\tau}' \quad p; \overline{\alpha <: N} \vdash \tau_0 <: [\overline{\alpha / \beta}] \tau'_0}{p; \Delta \vdash \text{override}(f, \{\overrightarrow{L}\}, [\overline{\alpha \text{ extends } N}]) \ \overrightarrow{\tau} \rightarrow \tau_0}$$

Method type lookup: $\boxed{\text{mtype}_p(f, \tau) = \{[\overline{\alpha \text{ extends } N}] \ \overrightarrow{\tau} \rightarrow \tau\}}$

$$[\text{MT-SELF}] \quad \frac{- C[\overline{\alpha \text{ extends } _}] _ \overrightarrow{fd} _ \in p \quad f[\overline{\beta \text{ extends } M}](\overline{_ : \tau'}) : \tau'_0 = e \in \{\overrightarrow{fd}\}}{\text{mtype}_p(f, C[\overline{\tau}]) = \{[\overline{\tau / \alpha}] [\overline{\beta \text{ extends } M}] \ \overrightarrow{\tau}' \rightarrow \tau'_0\}}$$

$$[\text{MT-SUPER}] \quad \frac{- C[\overline{\alpha \text{ extends } _}] _ \text{ extends } \{\overrightarrow{N}\} _ \overrightarrow{fd} _ \in p \quad f \notin \{\overline{Fname}(\overrightarrow{fd})\}}{\text{mtype}_p(f, C[\overline{\tau}]) = \bigcup_{N_i \in \{\overrightarrow{N}\}} \text{mtype}_p(f, [\overline{\tau / \alpha}] N_i)}$$

$$[\text{MT-OBJ}] \quad \text{mtype}_p(f, \text{Object}) = \emptyset$$

Figure A.3: Static Semantics of Basic Core Fortress (I)

Expression typing: $p; \Delta; \Gamma \vdash e : \tau$

$$\begin{array}{c}
\text{[T-VAR]} \quad p; \Delta; \Gamma \vdash x : \Gamma(x) \\
\text{[T-SELF]} \quad p; \Delta; \Gamma \vdash \mathbf{self} : \Gamma(\mathbf{self}) \\
\text{[T-OBJECT]} \quad \frac{\text{object } O[\overrightarrow{\alpha \text{ extends } _}] (\overrightarrow{_} : \tau') \text{ - end } \in p \quad p; \Delta \vdash O[\overrightarrow{\tau}] \text{ ok}}{p; \Delta; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau} \quad p; \Delta \vdash \overrightarrow{\tau} <: [\overrightarrow{\tau} / \overrightarrow{\alpha}] \overrightarrow{\tau}'}{p; \Delta; \Gamma \vdash O[\overrightarrow{\tau}](\overrightarrow{e}) : O[\overrightarrow{\tau}]} \\
\text{[T-FIELD]} \quad \frac{p; \Delta; \Gamma \vdash e_0 : \tau_0 \quad \text{bound}_\Delta(\tau_0) = O[\overrightarrow{\tau}]}{p; \Delta; \Gamma \vdash e_0.x_i : [\overrightarrow{\tau} / \overrightarrow{\alpha}] \tau_i} \quad \text{object } O[\overrightarrow{\alpha \text{ extends } _}] (\overrightarrow{x} : \overrightarrow{\tau}) \text{ - end } \in p \\
\text{[T-METHOD]} \quad \frac{p; \Delta; \Gamma \vdash e_0 : \tau_0 \quad \text{mtype}_p(f, \text{bound}_\Delta(\tau_0)) = \{[\overrightarrow{\alpha \text{ extends } \overrightarrow{N}}] \overrightarrow{\tau} \rightarrow \tau'_0\}}{p; \Delta; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau} \quad p; \Delta \vdash \overrightarrow{\tau} <: [\overrightarrow{\tau} / \overrightarrow{\alpha}] \overrightarrow{N}} \quad \frac{p; \Delta \vdash \overrightarrow{\tau} \text{ ok} \quad p; \Delta \vdash \overrightarrow{\tau} <: [\overrightarrow{\tau} / \overrightarrow{\alpha}] \overrightarrow{N}}{p; \Delta; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau} \quad p; \Delta \vdash \overrightarrow{\tau} <: [\overrightarrow{\tau} / \overrightarrow{\alpha}] \overrightarrow{\tau}'}{p; \Delta; \Gamma \vdash e_0.f[\overrightarrow{\tau}](\overrightarrow{e}) : [\overrightarrow{\tau} / \overrightarrow{\alpha}] \tau'_0}
\end{array}$$

Subtyping: $p; \Delta \vdash \tau <: \tau$

$$\begin{array}{c}
\text{[S-OBJ]} \quad p; \Delta \vdash \tau <: \text{Object} \\
\text{[S-REFL]} \quad p; \Delta \vdash \tau <: \tau \\
\text{[S-TRANS]} \quad \frac{p; \Delta \vdash \tau_1 <: \tau_2 \quad p; \Delta \vdash \tau_2 <: \tau_3}{p; \Delta \vdash \tau_1 <: \tau_3} \\
\text{[S-VAR]} \quad p; \Delta \vdash \alpha <: \Delta(\alpha) \\
\text{[S-TAPP]} \quad \frac{_ \text{ C}[\overrightarrow{\alpha \text{ extends } _}] \text{ - extends } \{\overrightarrow{N}\} \text{ - } \in p}{p; \Delta \vdash C[\overrightarrow{\tau}] <: [\overrightarrow{\tau} / \overrightarrow{\alpha}] N_i}
\end{array}$$

Well-formed types: $p; \Delta \vdash \tau \text{ ok}$

$$\begin{array}{c}
\text{[W-OBJ]} \quad p; \Delta \vdash \text{Object ok} \\
\text{[W-VAR]} \quad \frac{\alpha \in \text{dom}(\Delta)}{p; \Delta \vdash \alpha \text{ ok}} \\
\text{[W-TAPP]} \quad \frac{_ \text{ C}[\overrightarrow{\alpha \text{ extends } \overrightarrow{N}}] \text{ - } \in p \quad p; \Delta \vdash \overrightarrow{\tau} \text{ ok} \quad p; \Delta \vdash \overrightarrow{\tau} <: [\overrightarrow{\tau} / \overrightarrow{\alpha}] \overrightarrow{N}}{p; \Delta \vdash C[\overrightarrow{\tau}] \text{ ok}}
\end{array}$$

Figure A.4: Static Semantics of Basic Core Fortress (II)

Bound of type: $\boxed{bound_{\Delta}(\tau) = \sigma}$

$$\begin{aligned} bound_{\Delta}(\alpha) &= \Delta(\alpha) \\ bound_{\Delta}(\sigma) &= \sigma \end{aligned}$$

One owner for all the visible methods: $\boxed{p \vdash oneOwner(C)}$

$$[ONEOWNER] \quad \frac{\forall f \in visible_p(C) . f \text{ only occurs once in } visible_p(C)}{p \vdash oneOwner(C)}$$

Auxiliary functions for methods: $\boxed{defined_p / inherited_p / visible_p(C) = \{\vec{f}\}}$

$$defined_p(C) = \{\overrightarrow{Fname(fd)}\} \quad \text{where } _ C _ \vec{fd} _ \in p$$

$$inherited_p(C) = \biguplus_{N_i \in \{\vec{N}\}} \{f_i \mid f_i \in visible_p(N_i), f_i \notin defined_p(C)\} \quad \text{where } _ C _ \text{extends } \{\vec{N}\} _ \in p$$

$$visible_p(C) = defined_p(C) \uplus inherited_p(C)$$

Figure A.5: Static Semantics of Basic Core Fortress (III)

A.2 Core Fortress with Where Clauses

In this section, we define a Fortress core calculus with `where` clauses. We call this calculus *Core Fortress with Where Clauses*. Core Fortress with Where Clauses is an extension of Basic Core Fortress with `where` clauses.

A.2.1 Syntax

The syntax for Core Fortress with Where Clauses is provided in Figure A.6. For simplicity, we use the following notational conventions:

- We abbreviate a sequence of relations $\alpha_1 \text{ extends } \{\vec{K}_1\} \cdots \alpha_n \text{ extends } \{\vec{K}_n\}$ to $\alpha \text{ extends } \{\vec{K}\}$ and $\tau_1 <: H_{11} \tau_1 <: H_{12} \cdots \tau_1 <: H_{1l} \tau_2 <: H_{21} \cdots \tau_n <: H_{nm}$ to $\vec{\tau} <: \vec{H}$.
- Substitutions of x with v and α with τ are denoted as $[v/x]$ and $[\tau/\alpha]$, respectively. A sequence of substitutions represents the composition of those substitutions where the right-most substitution is applied first. For example, $S_n \cdots S_1 \tau$ represents $S_n(\cdots S_2(S_1 \tau) \cdots)$.

Most of the syntax is a straightforward extension of Basic Core Fortress in Section A.1.1. An object or trait definition may include `where` clauses. Every method invocation is annotated with three kinds of static types by type inference: the static types of the receiver, the arguments, and the result. These type annotations appear in Core Fortress with Where Clauses in a form of `as τ` . If the annotated types are not enough (to find “witnesses” for the where-clauses variables), type checking rejects the program and requires more type information from the programmer.

A.2.2 Dynamic Semantics

A dynamic semantics for Core Fortress with Where Clauses is provided in Figures A.7 and A.8.

α, β		type variables
m		method name
x		field name
T		trait name
O		object name
C		trait or object name
τ, τ', τ''	$::= \alpha$	type
	σ	
σ	$::= N$	type that is not a type variable
	$O[\vec{\tau}]$	
N, M, L	$::= T[\vec{\tau}]$	super trait
	Object	
K, H, J	$::= \alpha$	bound on a type variable
	N	
e	$::= x$	expression
	self	
	$O[\vec{\tau}](\vec{e})$	
	$e.x$	
	$(e \text{ as } C[\vec{\tau}]).m[\vec{\tau}](e \text{ as } \vec{\tau}) \text{ as } \tau$	
	typecase $x = e$ of $\vec{\tau} \Rightarrow \vec{e}$ else e end	
md	$::= m[\alpha \text{ extends } \{\vec{K}\}](x:\vec{\tau}):\tau = e$	method definition
td	$::= \text{trait } T[\alpha \text{ extends } \{\vec{K}\}] \text{ extends } \{\vec{N}\} \text{ where } \{\alpha \text{ extends } \{\vec{K}\}\} \vec{md} \text{ end}$	trait definition
od	$::= \text{object } O[\alpha \text{ extends } \{\vec{K}\}](x:\vec{\tau}) \text{ extends } \{\vec{N}\} \text{ where } \{\alpha \text{ extends } \{\vec{K}\}\} \vec{md} \text{ end}$	object definition
d	$::= td$	definition
	od	
p	$::= \vec{d} e$	program

Figure A.6: Syntax of Core Fortress with Where Clauses

A.2.3 Static Semantics

A static semantics for Core Fortress with Where Clauses is provided in Figures A.9, A.10, A.11, and A.12.

For simplicity, we use the following conventions:

- $FTV(\tau)$ collects all free type variables in τ .
- Similarly, $FTV(e)$ collects all free type variables in all types in e .

We proved the type soundness of Core Fortress with Where Clauses using the standard technique of proving a progress theorem and a subject reduction theorem.

Values, evaluation contexts, and redexes

$$\begin{aligned}
v &::= O[\vec{\tau}](\vec{v}) \\
E &::= \square \\
&| O[\vec{\tau}](\vec{v} E \vec{e}) \\
&| E.x \\
&| E \text{ as } \tau.m[\vec{\tau}](\vec{e} \text{ as } \vec{\tau}) \text{ as } \tau \\
&| v \text{ as } \tau.m[\vec{\tau}](\vec{v} \text{ as } \vec{\tau} E \text{ as } \tau \vec{e} \text{ as } \vec{\tau}) \text{ as } \tau \\
&| \text{typecase } x = E \text{ of } \vec{\tau} \Rightarrow \vec{e} \text{ else } e \text{ end} \\
R &::= v.x \\
&| v \text{ as } \tau.m[\vec{\tau}](\vec{v} \text{ as } \vec{\tau}) \text{ as } \tau \\
&| \text{typecase } x = v \text{ of } \vec{\tau} \Rightarrow \vec{e} \text{ else } e \text{ end}
\end{aligned}$$

Evaluation rules: $\boxed{p \vdash E[R] \longrightarrow E[e]}$

[R-FIELD]
$$\frac{\text{object } O[\overline{\alpha \text{ extends } _}](\vec{x}:\vec{_}) _ \text{ end} \in p}{p \vdash E[O[\vec{\tau}](\vec{v}).x_i] \longrightarrow E[v_i]}$$

[R-METHOD]
$$\frac{\begin{array}{l} mtype_p(m, C[\vec{\tau}^c], \emptyset) = \{([\alpha' \text{ extends } _]\vec{\tau}' \rightarrow \tau'_0, _)\} \quad S([\vec{\tau}'/\vec{\alpha}']\vec{\tau}') = \vec{\tau}^a \quad S([\vec{\tau}'/\vec{\alpha}']\tau'_0) = \tau^r \\ \text{object } O _ (\vec{x}:\vec{_}) _ \text{ end} \in p \quad mbody_p(m[\vec{\tau}'], O[\vec{\tau}'], C[\vec{\tau}^c]) = \{(x') \rightarrow e\} \end{array}}{p \vdash E[O[\vec{\tau}](\vec{v}) \text{ as } C[\vec{\tau}^c].m[\vec{\tau}'](\vec{v}' \text{ as } \tau^a) \text{ as } \tau^r] \longrightarrow E[[\vec{v}'/\vec{x}'] [O[\vec{\tau}](\vec{v})/\text{self}] [\vec{v}'/\vec{x}'] Se]}$$

[R-TYPECASE]
$$\frac{\text{type}(v) = \tau^v \quad |\vec{\tau}| = n \quad \neg(p; \emptyset \vdash \tau^v <: \tau_i) \quad 1 \leq i < n \quad p; \emptyset \vdash \tau^v <: \tau_n}{p \vdash E[\text{typecase } x = v \text{ of } \vec{\tau} \Rightarrow \vec{e} \vec{\tau}' \Rightarrow e' \text{ else } e \text{ end}] \longrightarrow E[[v/x]e_n]}$$

[R-TYPECASEELSE]
$$\frac{\text{type}(v) = \tau^v \quad \neg(p; \emptyset \vdash \tau^v <: \tau_i) \quad 1 \leq i \leq |\vec{\tau}|}{p \vdash E[\text{typecase } x = v \text{ of } \vec{\tau} \Rightarrow \vec{e} \text{ else } e' \text{ end}] \longrightarrow E[[v/x]e']}$$

Method body lookup: $\boxed{mbody_p(m[\vec{\tau}'], \tau, \tau) = \{(x') \rightarrow e\}}$

[MB-SELF]
$$\frac{_ C[\overline{\alpha \text{ extends } _}] _ \vec{md} _ \in p \quad m[\overline{\alpha' \text{ extends } _}](x':\vec{_}) _ = e \in \{\vec{md}\}}{mbody_p(m[\vec{\tau}'], C[\vec{\tau}'], C[\vec{\tau}^c]) = \{(x') \rightarrow [\vec{\tau}'/\vec{\alpha}'] [\vec{\tau}^c/\vec{\alpha}'] e\}}$$

[MB-WITNESS]
$$\frac{_ C[\overline{\alpha \text{ extends } _}] _ \vec{md} _ \in p \quad m[\overline{\alpha' \text{ extends } _}](x':\vec{_}) _ = e \in \{\vec{md}\} \quad \tau^o \neq C[_]_ \quad \text{witness}_p(C[\vec{\tau}'], \tau^o) = S}{mbody_p(m[\vec{\tau}'], C[\vec{\tau}'], \tau^o) = \{(x') \rightarrow [\vec{\tau}'/\vec{\alpha}'] (S([\vec{\tau}/\vec{\alpha}]e))\}}$$

[MB-SUPER]
$$\frac{_ C[\overline{\alpha \text{ extends } _}] _ \text{extends } \{\vec{N}\} _ \vec{md} _ \in p \quad m \notin \{\overline{\text{Name}(\vec{md})}\} \quad C \neq C'}{mbody_p(m[\vec{\tau}'], C[\vec{\tau}'], C'[\vec{\tau}^c]) = \bigcup_{N_i \in \{\vec{N}\}} mbody_p(m[\vec{\tau}'], [\vec{\tau}'/\vec{\alpha}'] N_i, C'[\vec{\tau}^c])}$$

[MB-STATIC]
$$\frac{_ C[\overline{\alpha \text{ extends } _}] _ \text{extends } \{\vec{N}\} _ \vec{md} _ \in p \quad m \notin \{\overline{\text{Name}(\vec{md})}\}}{mbody_p(m[\vec{\tau}'], C[\vec{\tau}'], C[\vec{\tau}^c]) = \bigcup_{N_i \in \{\vec{N}\}} mbody_p(m[\vec{\tau}'], [\vec{\tau}^c/\vec{\alpha}'] N_i, \text{Object})}$$

[MB-OBJ]
$$mbody_p(m[\vec{\tau}'], \text{Object}, \tau) = \emptyset$$

Figure A.7: Dynamic Semantics of Core Fortress with Where Clauses (I)

Function/method name lookup: $\boxed{Name(md) = m}$

$$Name(m \overrightarrow{\alpha \text{ extends } \{K\}}(x:\vec{\tau}) : \tau = e) = m$$

Types of values: $\boxed{type(v) = \tau}$

$$type(O[\vec{\tau}](\vec{v})) = O[\vec{\tau}]$$

Finding witnesses from the static type of the receiver: $\boxed{witness_p(\tau, \tau) = S}$

$$witness_p(\tau, \tau') = \begin{cases} [] & \text{if } \tau' = \text{Object} \\ match(C[\vec{\tau}], C[\vec{\tau}']) & \text{if } \tau = C[\vec{\tau}] \text{ and } \tau' = C[\vec{\tau}'] \\ S_n \cdots S_1 & \text{if } \tau = C[\vec{\tau}], \tau' = C'[\vec{\tau}'], C \neq C', \\ & \text{-- } C[\overrightarrow{\alpha \text{ extends } _}] \text{ -- extends } \{\vec{N}\} \text{ -- } \in p, \\ & \text{and } witness_p([\vec{\tau}/\vec{\alpha}]N_i, C'[\vec{\tau}']) = S_i \text{ for } N_i \in \{\vec{N}\} \\ [] & \text{otherwise} \end{cases}$$

Match two types to get substitutions: $\boxed{match(\tau, \tau) = S}$

$$match(\tau, \tau') = \begin{cases} [] & \text{if } \tau = \tau' \\ [\tau'/\beta] & \text{if } \tau = \beta \\ S_n \cdots S_1 & \text{if } \tau = C[\vec{\tau}], \tau' = C'[\vec{\tau}'], \text{ and } match(S_{i-1} \cdots S_1 \tau_i, \tau'_i) = S_i \text{ for } 1 \leq i \leq n \\ [] & \text{otherwise} \end{cases}$$

Figure A.8: Dynamic Semantics of Core Fortress with Where Clauses (II)

Environments and method types

$$\begin{aligned} \Delta &::= \overrightarrow{\alpha <: \{\vec{K}\}} && \text{bound environment} \\ \Gamma &::= \overrightarrow{x : \vec{\tau}} && \text{type environment} \\ \eta &::= \llbracket \alpha \text{ extends } \{\vec{K}\} \rrbracket \overrightarrow{\vec{\tau}} \rightarrow \tau && \text{method type} \end{aligned}$$

Program typing: $\boxed{\vdash p : \tau}$

$$[\text{T-PROGRAM}] \quad \frac{p = \vec{d} \ e \quad p \vdash \vec{d} \text{ ok} \quad p; \emptyset; \emptyset \vdash e : \tau}{\vdash p : \tau}$$

Definition typing: $\boxed{p \vdash d \text{ ok}}$

$$[\text{T-TRAITDEF}] \quad \frac{\begin{array}{c} p \vdash \text{validMultipleInheritance}(T) \quad \Delta = \overrightarrow{\alpha <: \{\vec{K}\}} \beta <: \{\vec{H}\} \\ p; \Delta \vdash \vec{K} \text{ ok} \quad p; \Delta \vdash \vec{N} \text{ ok} \quad p; \Delta \vdash \vec{H} \text{ ok} \quad p; \Delta; \text{self} : T[\overrightarrow{\alpha}]; T \vdash \vec{m}d \text{ ok} \end{array}}{p \vdash \text{trait } T \llbracket \alpha \text{ extends } \{\vec{K}\} \rrbracket \text{ extends } \{\vec{N}\} \text{ where } \{\beta \text{ extends } \{\vec{H}\}\} \vec{m}d \text{ end ok}}$$

$$[\text{T-OBJECTDEF}] \quad \frac{\begin{array}{c} p \vdash \text{validMultipleInheritance}(O) \quad \Delta = \overrightarrow{\alpha <: \{\vec{K}\}} \quad \Delta' = \Delta \beta <: \{\vec{H}\} \\ p; \Delta' \vdash \vec{K} \text{ ok} \quad p; \Delta \vdash \vec{\tau} \text{ ok} \quad p; \Delta' \vdash \vec{N} \text{ ok} \quad p; \Delta' \vdash \vec{H} \text{ ok} \\ p; \Delta'; \text{self} : O[\overrightarrow{\alpha}] \overrightarrow{x : \vec{\tau}}; O \vdash \vec{m}d \text{ ok} \end{array}}{p \vdash \text{object } O \llbracket \alpha \text{ extends } \{\vec{K}\} \rrbracket (\overrightarrow{x : \vec{\tau}}) \text{ extends } \{\vec{N}\} \text{ where } \{\beta \text{ extends } \{\vec{H}\}\} \vec{m}d \text{ end ok}}$$

Method typing: $\boxed{p; \Delta; \Gamma; C \vdash md \text{ ok}}$

$$[\text{T-METHODDEF}] \quad \frac{\begin{array}{c} p \vdash \text{override}(m, C, \llbracket \alpha \text{ extends } \{\vec{K}\} \rrbracket \overrightarrow{\vec{\tau}} \rightarrow \tau_0) \quad \Delta' = \Delta \alpha <: \{\vec{K}\} \quad p; \Delta' \vdash \vec{K} \text{ ok} \\ p; \Delta' \vdash \vec{\tau} \text{ ok} \quad p; \Delta' \vdash \tau_0 \text{ ok} \quad p; \Delta'; \Gamma \overrightarrow{x : \vec{\tau}} \vdash e : \tau' \quad p; \Delta' \vdash \tau' <: \tau_0 \end{array}}{p; \Delta; \Gamma; C \vdash m \llbracket \alpha \text{ extends } \{\vec{K}\} \rrbracket (\overrightarrow{x : \vec{\tau}}) : \tau_0 = e \text{ ok}}$$

Method overriding: $\boxed{p \vdash \text{override}(m, C, \eta)}$

$$[\text{OVERRIDE}] \quad \frac{\begin{array}{c} - C \llbracket \alpha'' \text{ extends } \{\vec{K}''\} \rrbracket - \text{extends } \{\vec{N}\} \text{ where } \{\beta \text{ extends } \{\vec{H}\}\} - \in p \\ \Delta = \overrightarrow{\alpha'' <: \{\vec{K}''\}} \beta <: \{\vec{H}\} \\ \bigcup_{C' \llbracket \vec{\tau}'' \rrbracket \in \{\vec{N}\}} \text{mtype}_p(m, C' \llbracket \vec{\tau}'' \rrbracket, \Delta) = \{(\llbracket \alpha' \text{ extends } \{\vec{K}'\} \rrbracket \overrightarrow{\vec{\tau}}' \rightarrow \tau'_0, \Delta')\} \\ \vec{K} = [\overrightarrow{\alpha} / \overrightarrow{\alpha'}] \vec{K}' \quad p; \Delta' \vdash [\overrightarrow{\alpha} / \overrightarrow{\alpha'}] \overrightarrow{\vec{\tau}}' <: \overrightarrow{\vec{\tau}} \quad p; \Delta' \vdash \tau'_0 <: [\overrightarrow{\alpha} / \overrightarrow{\alpha'}] \tau'_0 \end{array}}{p \vdash \text{override}(m, C, \llbracket \alpha \text{ extends } \{\vec{K}\} \rrbracket \overrightarrow{\vec{\tau}} \rightarrow \tau_0)}$$

Valid multiple inheritance: $\boxed{p \vdash \text{validMultipleInheritance}(C)}$

$$[\text{VALIDMI}] \quad \frac{p \vdash \text{oneOwner}(C) \quad p \vdash \text{validWhere}(C)}{p \vdash \text{validMultipleInheritance}(C)}$$

Figure A.9: Static Semantics of Core Fortress with Where Clauses (I)

One owner for all the visible methods: $\boxed{p \vdash \text{oneOwner}(C)}$

$$\text{[ONEOWNER]} \quad \frac{\forall m \in \text{visible}_p(C) . f \text{ only occurs once in } \text{visible}_p(C)}{p \vdash \text{oneOwner}(C)}$$

Valid where clauses: $\boxed{p \vdash \text{validWhere}(C)}$

$$\begin{array}{l} \forall m \in \text{visible}_p(C) . \\ \text{where } _ C[\overrightarrow{\alpha}] _ \text{ extends } \{ \overrightarrow{N} \} _ \in p \\ \text{mbody}_p(m[\overrightarrow{\alpha}^f], C[\overrightarrow{\alpha}], C[\overrightarrow{\alpha}]) = \{ _ \rightarrow e_f \}, \quad \text{mtype}_p(m, C[\overrightarrow{\alpha}], \emptyset) = \{(\eta_f, \Delta)\} \\ 1. \quad \forall \beta \in (FTV(e_f) \setminus \{\overrightarrow{\alpha} \overrightarrow{\alpha}^f\}) . \beta \in FTV(\eta_f) \\ 2. \quad \forall \beta \in (FTV(\eta_f) \setminus \{\overrightarrow{\alpha} \overrightarrow{\alpha}^f\}) . \forall C'[\overrightarrow{\tau}^e] \in \bigcup_{N_i \in \{\overrightarrow{N}\}} \text{defining}_p(m, N_i) . \\ \beta = \tau_i^c \quad \Delta(\beta) = \overrightarrow{K}_i' \quad \text{for } 1 \leq i \leq |\overrightarrow{\tau}^e| \text{ where } _ C'[\overrightarrow{\alpha}' \text{ extends } \{\overrightarrow{K}'\}] _ \in p \end{array}$$

$$\text{[VALIDWHERE]} \quad \frac{}{p \vdash \text{validWhere}(C)}$$

Valid witnesses: $\boxed{p \vdash \text{validWitness}(\Delta, \overrightarrow{\alpha} <: \{\overrightarrow{\tau}\}, \overrightarrow{\tau})}$

$$\text{[VALIDWITNESS]} \quad \frac{\begin{array}{l} p; \Delta \vdash [\overrightarrow{\tau}^\beta / \overrightarrow{\beta}] \overrightarrow{\tau} \text{ ok} \quad p; \Delta \vdash \overrightarrow{\tau}^\beta \text{ ok} \quad p; \Delta \vdash \overrightarrow{\tau}^\beta <: [\overrightarrow{\tau}^\beta / \overrightarrow{\beta}] \overrightarrow{\tau} \\ \{\overrightarrow{\beta}\} \cap \text{dom}(\Delta) = \emptyset \end{array}}{p \vdash \text{validWitness}(\Delta, \overrightarrow{\beta} <: \{\overrightarrow{\tau}\}, \overrightarrow{\tau}^\beta)}$$

Expression typing: $\boxed{p; \Delta; \Gamma \vdash e : \tau}$

$$\text{[T-VAR]} \quad p; \Delta; \Gamma \vdash x : \Gamma(x)$$

$$\text{[T-SELF]} \quad p; \Delta; \Gamma \vdash \text{self} : \Gamma(\text{self})$$

$$\text{[T-OBJECT]} \quad \frac{\begin{array}{l} \text{object } O[\overrightarrow{\alpha} \text{ extends } _] (_ : \overrightarrow{\tau}') _ \text{ end} \in p \quad p; \Delta \vdash O[\overrightarrow{\tau}] \text{ ok} \\ p; \Delta; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau}'' \quad p; \Delta \vdash \overrightarrow{\tau}'' <: [\overrightarrow{\tau} / \overrightarrow{\alpha}] \overrightarrow{\tau}' \end{array}}{p; \Delta; \Gamma \vdash O[\overrightarrow{\tau}](\overrightarrow{e}) : O[\overrightarrow{\tau}]}$$

$$\text{[T-FIELD]} \quad \frac{p; \Delta; \Gamma \vdash e_0 : \tau_0 \quad \text{bound}_\Delta(\tau_0) = \{O[\overrightarrow{\tau}]\} \quad \text{object } O[\overrightarrow{\alpha} \text{ extends } _](x:\overrightarrow{\tau}) _ \text{ end} \in p}{p; \Delta; \Gamma \vdash e_0 . x_i : [\overrightarrow{\tau} / \overrightarrow{\alpha}] \tau_i}$$

$$\text{[T-METHOD]} \quad \frac{\begin{array}{l} p; \Delta; \Gamma \vdash e_0 : \tau_0 \quad p; \Delta \vdash \tau_0 <: C[\overrightarrow{\tau}^c] \quad \text{mtype}_p(m, C[\overrightarrow{\tau}^c], \emptyset) = \{([\overrightarrow{\alpha}' \text{ extends } \{\overrightarrow{K}'\}] \overrightarrow{\tau}' \rightarrow \tau_0', \Delta')\} \\ p; \Delta \vdash \overrightarrow{\tau} \text{ ok} \quad p; \Delta \vdash C[\overrightarrow{\tau}^c] \text{ ok} \quad p; \Delta \vdash \overrightarrow{\tau}^a \text{ ok} \quad p; \Delta \vdash \tau^r \text{ ok} \\ p; \Delta; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau}'' \quad p; \Delta \vdash \overrightarrow{\tau}'' <: \overrightarrow{\tau}^a \quad \text{dom}(\Delta') = \{\overrightarrow{\beta}\} \quad S = [\overrightarrow{\tau}^\beta / \overrightarrow{\beta}] \\ p \vdash \text{validWitness}(\Delta, \Delta', \overrightarrow{\tau}^\beta) \quad p; \Delta \vdash \overrightarrow{\tau} <: S([\overrightarrow{\tau} / \overrightarrow{\alpha}] \overrightarrow{K}') \quad S([\overrightarrow{\tau} / \overrightarrow{\alpha}] \overrightarrow{\tau}') = \overrightarrow{\tau}^a \quad S([\overrightarrow{\tau} / \overrightarrow{\alpha}] \tau_0') = \tau^r \end{array}}{p; \Delta; \Gamma \vdash e_0 \text{ as } C[\overrightarrow{\tau}^c] . m[\overrightarrow{\tau}](e \text{ as } \overrightarrow{\tau}^a) \text{ as } \tau^r : \tau^r}$$

$$\text{[T-TYPECASE]} \quad \frac{\begin{array}{l} p; \Delta; \Gamma \vdash e : \tau \quad p; \Delta; \Gamma \quad x : \tau_i \vdash e_i : \tau_i^e \quad p; \Delta \vdash \tau_i^e <: \tau' \quad 1 \leq i \leq |\overrightarrow{\tau}| \\ p; \Delta; \Gamma \quad x : \tau \vdash e' : \tau^{e'} \quad p; \Delta \vdash \tau^{e'} <: \tau' \end{array}}{p; \Delta; \Gamma \vdash \text{typecase } x = e \text{ of } \overrightarrow{\tau} \Rightarrow \overrightarrow{e} \text{ else } e' \text{ end} : \tau'}$$

Figure A.10: Static Semantics of Core Fortress with Where Clauses (II)

Subtyping: $\boxed{p; \Delta \vdash \tau <: \tau}$

[S-OBJ] $p; \Delta \vdash \tau <: \text{Object}$

[S-REFL] $p; \Delta \vdash \tau <: \tau$

[S-TRANS]
$$\frac{p; \Delta \vdash \tau_1 <: \tau_2 \quad p; \Delta \vdash \tau_2 <: \tau_3}{p; \Delta \vdash \tau_1 <: \tau_3}$$

[S-VAR]
$$\frac{\tau \in \Delta(\alpha)}{p; \Delta \vdash \alpha <: \tau}$$

[S-TAPP]
$$\frac{p; \Delta \vdash \vec{\tau} \text{ ok} \quad \overrightarrow{- C[\alpha \text{ extends } \{\vec{K}\}] \text{ extends } \{\vec{N}\} \text{ where } \{\beta \text{ extends } \{\vec{H}\}\} - \in p} \quad p; \Delta \vdash \vec{\tau} <: [\vec{\tau}^\beta / \vec{\beta}][\vec{\tau} / \vec{\alpha}] \vec{K} \quad p \vdash \text{validWitness}(\Delta, \beta <: \{[\vec{\tau} / \vec{\alpha}] \vec{H}\}, \tau^\beta)}{p; \Delta \vdash C[\vec{\tau}] <: [\vec{\tau}^\beta / \vec{\beta}][\vec{\tau} / \vec{\alpha}] N_i}$$

Well-formed types: $\boxed{p; \Delta \vdash \tau \text{ ok}}$

[W-OBJ] $p; \Delta \vdash \text{Object ok}$

[W-VAR]
$$\frac{\alpha \in \text{dom}(\Delta)}{p; \Delta \vdash \alpha \text{ ok}}$$

[W-TAPP]
$$\frac{p; \Delta \vdash \vec{\tau} \text{ ok} \quad \overrightarrow{- C[\alpha \text{ extends } \{\vec{K}\}] - \text{ where } \{\beta \text{ extends } \{\vec{H}\}\} - \in p} \quad p; \Delta \vdash \vec{\tau} <: [\vec{\tau}^\beta / \vec{\beta}][\vec{\tau} / \vec{\alpha}] \vec{K} \quad p \vdash \text{validWitness}(\Delta, \beta <: \{[\vec{\tau} / \vec{\alpha}] \vec{H}\}, \tau^\beta)}{p; \Delta \vdash C[\vec{\tau}] \text{ ok}}$$

Method type lookup: $\boxed{mtype_p(m, \tau, \Delta) = \{(\eta, \Delta)\}}$

[MT-SELF]
$$\frac{\overrightarrow{- C[\alpha \text{ extends } _] - \text{ where } \{\beta \text{ extends } \{\vec{H}\}\} - \vec{md} - \in p} \quad m[\alpha' \text{ extends } \{\vec{K}'\}](_ : \tau') : \tau_0 - \in \{\vec{md}\}}{\Delta' = \Delta \beta <: \{\vec{H}\}}}{mtype_p(m, C[\vec{\tau}], \Delta) = \{([\vec{\tau} / \vec{\alpha}][\alpha' \text{ extends } \{\vec{K}'\}] \vec{\tau}' \rightarrow \tau_0, [\vec{\tau} / \vec{\alpha}] \Delta')\}}$$

[MT-SUPER]
$$\frac{\overrightarrow{- C[\alpha \text{ extends } _] \text{ extends } \{\vec{N}\} \text{ where } \{\beta \text{ extends } \{\vec{H}\}\} - \vec{md} - \in p} \quad m \notin \{\text{Name}(\vec{md})\}}{\Delta' = \Delta \beta <: \{\vec{H}\}}}{mtype_p(m, C[\vec{\tau}], \Delta) = \bigcup_{N_i \in \{\vec{N}\}} mtype_p(m, [\vec{\tau} / \vec{\alpha}] N_i, [\vec{\tau} / \vec{\alpha}] \Delta')}$$

[MT-OBJ] $mtype_p(m, \text{Object}, \Delta) = \emptyset$

Bound of type: $\boxed{bound_\Delta(\tau) = \{\vec{\sigma}\}}$

$bound_\Delta(\alpha) = \bigcup_{\tau_i \in \Delta(\alpha)} bound_\Delta(\tau_i)$
 $bound_\Delta(\sigma) = \{\sigma\}$

Figure A.11: Static Semantics of Core Fortress with Where Clauses (III)

Traits defining a method: $\boxed{\text{defining}_p(m, N) = \{\vec{N}\}}$

$$\text{defining}_p(m, \text{Object}) = \emptyset$$

$$\text{defining}_p(m, C[\vec{\tau}]) = \begin{cases} \bigcup_{N_i \in \{\vec{N}\}} \text{defining}_p(m, [\vec{\tau}/\vec{\alpha}]N_i) & \text{if } _ C[\vec{\alpha}] _ \text{ extends } \{\vec{N}\} _ \in p \text{ and } m \notin \text{defined}_p(C) \\ \bigcup_{N_i \in \{\vec{N}\}} \text{defining}_p(m, [\vec{\tau}/\vec{\alpha}]N_i) \cup \{C[\vec{\tau}]\} & \text{if } _ C[\vec{\alpha}] _ \text{ extends } \{\vec{N}\} _ \in p \text{ and } m \in \text{defined}_p(C) \end{cases}$$

Auxiliary functions for methods: $\boxed{\text{defined}_p / \text{inherited}_p / \text{visible}_p(C) = \{\vec{m}\}}$

$$\text{defined}_p(C) = \{\overrightarrow{\text{Name}(md)}\} \quad \text{where } _ C _ \overrightarrow{md} _ \in p$$

$$\text{inherited}_p(C) = \bigsqcup_{N_i \in \{\vec{N}\}} \{m_i \mid m_i \in \text{visible}_p(N_i), m_i \notin \text{defined}_p(C)\} \quad \text{where } _ C _ \text{ extends } \{\vec{N}\} _ \in p$$

$$\text{visible}_p(C) = \text{defined}_p(C) \sqcup \text{inherited}_p(C)$$

Figure A.12: Static Semantics of Core Fortress with Where Clauses (IV)

A.3 Core Fortress with Overloading

In this section, we define a Fortress core calculus with overloading for dotted methods and first-order functions. We call this calculus *Core Fortress with Overloading*. Core Fortress with Overloading is an extension of Basic Core Fortress with overloading.

A.3.1 Syntax

The syntax for Core Fortress with Overloading is provided in Figure A.13.

A.3.2 Dynamic Semantics

A dynamic semantics for Core Fortress with Overloading is provided in Figure A.14.

A.3.3 Static Semantics

A static semantics for Core Fortress with Overloading is provided in Figures A.15, A.16, A.17, and A.18.

We proved the type soundness of Core Fortress with Overloading using the standard technique of proving a progress theorem and a subject reduction theorem.

α, β		type variables
m		function or method name
x		field name
T		trait name
O		object name
τ, τ', τ''	$::= \alpha$	type
	σ	
σ	$::= N$	type that is not a type variable
	$O[\vec{\tau}]$	
N, M, L	$::= T[\vec{\tau}]$	type that can be a type bound
	Object	
e	$::= x$	expression
	self	
	$O[\vec{\tau}](\vec{e})$	
	$e.x$	
	$e.m[\vec{\tau}](\vec{e})$	
	$m[\vec{\tau}](\vec{e})$	
md	$::= m[\overrightarrow{\alpha \text{ extends } N}](\overrightarrow{x:\vec{\tau}}): \tau = e$	function or method definition
td	$::= \text{trait } T[\overrightarrow{\alpha \text{ extends } N}] \text{ extends } \{\overrightarrow{N}\} \overrightarrow{md} \text{ end}$	trait definition
od	$::= \text{object } O[\overrightarrow{\alpha \text{ extends } N}](\overrightarrow{x:\vec{\tau}}) \text{ extends } \{\overrightarrow{N}\} \overrightarrow{md} \text{ end}$	object definition
d	$::= md$	definition
	td	
	od	
p	$::= \overrightarrow{d} e$	program

Figure A.13: Syntax of Core Fortress with Overloading

Values, evaluation contexts, and redexes

$$\begin{array}{ll}
v ::= O[\vec{\tau}](\vec{v}) & \text{value} \\
E ::= \square & \text{evaluation context} \\
| O[\vec{\tau}](\vec{e} E \vec{e}') & \\
| E.x & \\
| E.m[\vec{\tau}](\vec{e}') & \\
| e.m[\vec{\tau}](\vec{e} E \vec{e}') & \\
| m[\vec{\tau}](\vec{e} E \vec{e}') & \\
R ::= v.x & \text{redex} \\
| v.m[\vec{\tau}](\vec{v}) & \\
| m[\vec{\tau}](\vec{v}) &
\end{array}$$

Evaluation rules: $\boxed{p \vdash E[R] \longrightarrow E[e]}$

[R-FIELD]
$$\frac{\text{object } O_{-}(\vec{x}:\vec{-})_{-} \text{ end} \in p}{p \vdash E[O[\vec{\tau}](\vec{v}).x_i] \longrightarrow E[v_i]}$$

[R-METHOD]
$$\frac{\text{object } O[\overline{\alpha \text{ extends } N}](\vec{x}:\vec{-})_{-} \text{ end} \in p \quad \overline{\text{type}(v')} = \vec{\tau}''}{\text{mostspecific}_{p;\emptyset}(\text{applicable}_{p;\emptyset}(m[\vec{\tau}'](\vec{\tau}''), \text{visible}_p(O[\vec{\tau}])) = m[\overline{\alpha' \text{ extends } N'}](\vec{x}':\vec{-})_{-} = e)}{p \vdash E[O[\vec{\tau}](\vec{v}).m[\vec{\tau}'](\vec{v}')] \longrightarrow E[\vec{v}/\vec{x}][O[\vec{\tau}](\vec{v})/\text{self}][\vec{v}'/\vec{x}']e]}$$

[R-FUNCTION]
$$\frac{\overline{\text{type}(v)} = \vec{\tau}' \quad \text{mostspecific}_{p;\emptyset}(\text{applicable}_{p;\emptyset}(m[\vec{\tau}](\vec{\tau}'), \{(md, \text{Object}) \mid md \in p\}) = m[\overline{\alpha \text{ extends } N}](\vec{x}:\vec{-})_{-} = e)}{p \vdash E[m[\vec{\tau}](\vec{v})] \longrightarrow E[\vec{v}/\vec{x}]e]}$$

Types of values: $\boxed{\text{type}(v) = \tau}$

$\text{type}(O[\vec{\tau}](\vec{v})) = O[\vec{\tau}]$

Figure A.14: Dynamic Semantics of Core Fortress with Overloading

Environments and trait or object names

Δ	::= $\overrightarrow{\alpha <: N}$	bound environment
Γ	::= $\overrightarrow{x : \tau}$	type environment
C	::= T	trait name
	O	object name

Program typing: $\boxed{\vdash p : \tau}$

$$\text{[T-PROGRAM]} \quad \frac{p = \overrightarrow{d} \ e \quad p; \emptyset; \emptyset \vdash \overrightarrow{d} \text{ ok} \quad p \vdash \text{validFun}(\overrightarrow{d}) \quad p; \emptyset; \emptyset \vdash e : \tau}{\vdash p : \tau}$$

Trait typing: $\boxed{p; \emptyset; \emptyset \vdash td \text{ ok}}$

$$\text{[T-TRAITDEF]} \quad \frac{\Delta = \overrightarrow{\alpha <: N} \quad p; \Delta \vdash \overrightarrow{N} \text{ ok} \quad p; \Delta \vdash \overrightarrow{M} \text{ ok} \quad p; \Delta; \text{self} : T[\overrightarrow{\alpha}] \vdash \overrightarrow{md} \text{ ok} \quad p \vdash \text{validMeth}(T)}{p; \emptyset; \emptyset \vdash \text{trait } T[\overrightarrow{\alpha} \text{ extends } \overrightarrow{N}] \text{ extends } \{\overrightarrow{M}\} \overrightarrow{md} \text{ end ok}}$$

Object typing: $\boxed{p; \emptyset; \emptyset \vdash od \text{ ok}}$

$$\text{[T-OBJECTDEF]} \quad \frac{\Delta = \overrightarrow{\alpha <: N} \quad p; \Delta \vdash \overrightarrow{N} \text{ ok} \quad p; \Delta \vdash \overrightarrow{\tau} \text{ ok} \quad p; \Delta \vdash \overrightarrow{M} \text{ ok} \quad p; \Delta; \text{self} : O[\overrightarrow{\alpha}] \overrightarrow{x : \tau} \vdash \overrightarrow{md} \text{ ok} \quad p \vdash \text{validMeth}(O)}{p; \emptyset; \emptyset \vdash \text{object } O[\overrightarrow{\alpha} \text{ extends } \overrightarrow{N}] (\overrightarrow{x : \tau}) \text{ extends } \{\overrightarrow{M}\} \overrightarrow{md} \text{ end ok}}$$

Function and method typing: $\boxed{p; \Delta; \Gamma \vdash md \text{ ok}}$

$$\text{[T-FUNMETHDEF]} \quad \frac{\Delta' = \Delta \ \alpha <: \{\overrightarrow{N}\} \quad p; \Delta' \vdash \overrightarrow{N} \text{ ok} \quad p; \Delta' \vdash \overrightarrow{\tau} \text{ ok} \quad p; \Delta' \vdash \tau_0 \text{ ok} \quad p; \Delta'; \Gamma \overrightarrow{x : \tau} \vdash e : \tau' \quad p; \Delta' \vdash \tau' <: \tau_0}{p; \Delta; \Gamma \vdash m[\overrightarrow{\alpha} \text{ extends } \overrightarrow{N}] (\overrightarrow{x : \tau}) : \tau_0 = e \text{ ok}}$$

Valid method declarations: $\boxed{p \vdash \text{validMeth}(C)}$

$$\begin{aligned} & \forall (md, C[\overrightarrow{\tau^c}]), (md', C'[\overrightarrow{\tau^{c'}}]) \in \text{visible}_p(C^o[\overrightarrow{\alpha^o}]). \\ & \text{where } _ \ C^o[\overrightarrow{\alpha^o} \text{ extends } _] _ \in p, \\ & \quad md \neq md' \quad (\text{not same declaration}), \\ & \quad md = m[\overrightarrow{\alpha} \text{ extends } \overrightarrow{N}] (_ : \overrightarrow{\tau}) : \tau^r = _, \quad md' = m[\overrightarrow{\alpha'} \text{ extends } \overrightarrow{N'}] (_ : \overrightarrow{\tau'}) : \tau^{r'} = _, \\ \text{[VALIDMETH]} & \quad \frac{p \vdash \text{valid}([\overrightarrow{\alpha} \text{ extends } \overrightarrow{N}] C[\overrightarrow{\tau^c}] \overrightarrow{\tau} \rightarrow \tau^r, [\overrightarrow{\alpha'} \text{ extends } \overrightarrow{N'}] C'[\overrightarrow{\tau^{c'}}] \overrightarrow{\tau'} \rightarrow \tau^{r'}, \text{visible}_p(C^o[\overrightarrow{\alpha^o}]))}{p \vdash \text{validMeth}(C^o)} \end{aligned}$$

Figure A.15: Static Semantics of Core Fortress with Overloading (I)

Valid function declarations: $p \vdash \text{validFun}(\vec{d})$

$\forall md, md' \in \vec{d}.$

where $md \neq md'$ (not same declaration),

$md = m[\overrightarrow{\alpha \text{ extends } N}](\overrightarrow{-:\vec{\tau}}):\tau^r = -, \quad md' = m[\overrightarrow{\alpha' \text{ extends } N'}](\overrightarrow{-:\vec{\tau}'}):\tau^{r'} = -,$

[VALIDFUN] $\frac{p \vdash \text{valid}([\overrightarrow{\alpha \text{ extends } N}]\text{Object } \vec{\tau} \rightarrow \tau^r, [\overrightarrow{\alpha' \text{ extends } N'}]\text{Object } \vec{\tau}' \rightarrow \tau^{r'}, \{(md, \text{Object}) \mid md \in \vec{d}\})}{p \vdash \text{validFun}(\vec{d})}$

Valid declarations: $p \vdash \text{valid}([\overrightarrow{\alpha \text{ extends } N}]\vec{\tau} \rightarrow \tau, [\overrightarrow{\alpha' \text{ extends } N'}]\vec{\tau}' \rightarrow \tau', \{(md, \tau)\})$

$\Delta = \overrightarrow{\alpha <: \{N\}}, \quad |\vec{\tau}| = n$

1. $|\vec{\tau}| \neq |\vec{\tau}'|$

∨ 2. 1) $\vec{N} = [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\vec{N}'$

∧ 2) $\forall 1 \leq i \leq n. p; \Delta \vdash \tau_i <: [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\tau'_i \quad \vee \quad p; \Delta \vdash [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\tau'_i <: \tau_i$

∧ 3) $\exists 1 \leq i \leq n. \tau_i \neq [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\tau'_i$

∧ 4) $\exists (m[\overrightarrow{\alpha'' \text{ extends } N''}](\overrightarrow{-:\vec{\tau}''}):\tau^{r''} = -, \tau_0'') \in S.$

where

$\forall 0 \leq i \leq n. p; \Delta \vdash \text{meet}(\{\tau_i, [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\tau'_i, [\overrightarrow{\alpha}/\overrightarrow{\alpha''}]\tau''_i\})$

∧ $\vec{N} = [\overrightarrow{\alpha}/\overrightarrow{\alpha''}]\vec{N}''$

∧ $p; \Delta \vdash [\overrightarrow{\alpha}/\overrightarrow{\alpha''}]\tau^{r''} <: \tau^r$

∧ $p; \Delta \vdash [\overrightarrow{\alpha'}/\overrightarrow{\alpha''}]\tau^{r''} <: \tau^{r'}$

[VALID] $\frac{\Delta = \overrightarrow{\alpha <: \{N\}}, \quad |\vec{\tau}| = n, \quad \begin{array}{l} 1. \quad |\vec{\tau}| \neq |\vec{\tau}'| \\ \vee 2. \quad 1) \vec{N} = [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\vec{N}' \\ \wedge 2) \forall 1 \leq i \leq n. p; \Delta \vdash \tau_i <: [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\tau'_i \quad \vee \quad p; \Delta \vdash [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\tau'_i <: \tau_i \\ \wedge 3) \exists 1 \leq i \leq n. \tau_i \neq [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\tau'_i \\ \wedge 4) \exists (m[\overrightarrow{\alpha'' \text{ extends } N''}](\overrightarrow{-:\vec{\tau}''}):\tau^{r''} = -, \tau_0'') \in S. \end{array}}{p \vdash \text{valid}([\overrightarrow{\alpha \text{ extends } N}]\vec{\tau} \rightarrow \tau^r, [\overrightarrow{\alpha' \text{ extends } N'}]\vec{\tau}' \rightarrow \tau^{r'}, S)}$

Expression typing: $p; \Delta; \Gamma \vdash e : \tau$

[T-VAR] $p; \Delta; \Gamma \vdash x : \Gamma(x)$

[T-SELF] $p; \Delta; \Gamma \vdash \text{self} : \Gamma(\text{self})$

[T-OBJECT] $\frac{\text{object } O[\overrightarrow{\alpha \text{ extends } -}](\overrightarrow{-:\vec{\tau}'}) \text{ end} \in p \quad p; \Delta \vdash O[\vec{\tau}] \text{ ok} \quad p; \Delta; \Gamma \vdash \vec{e} : \vec{\tau}'' \quad p; \Delta \vdash \tau'' <: [\vec{\tau}/\vec{\alpha}]\vec{\tau}'}{p; \Delta; \Gamma \vdash O[\vec{\tau}](\vec{e}) : O[\vec{\tau}]}$

[T-FIELD] $\frac{p; \Delta; \Gamma \vdash e_0 : \tau_0 \quad \text{bound}_\Delta(\tau_0) = O[\vec{\tau}'] \quad \text{object } O[\overrightarrow{\alpha \text{ extends } -}](\vec{x}:\vec{\tau}) \text{ end} \in p}{p; \Delta; \Gamma \vdash e_0 . x_i : [\vec{\tau}'/\vec{\alpha}]\tau_i}$

[T-METHOD] $\frac{p; \Delta; \Gamma \vdash e_0 : \tau_0 \quad p; \Delta \vdash \vec{\tau} \text{ ok} \quad p; \Delta; \Gamma \vdash \vec{e} : \vec{\tau}' \quad \text{mostspecific}_{p;\Delta}(\text{applicable}_{p;\Delta}(m[\vec{\tau}](\vec{\tau}'), \text{visible}_p(\text{bound}_\Delta(\tau_0)))) = m[\overrightarrow{\alpha \text{ extends } N}](\overrightarrow{-:\vec{\tau}^r})}{p; \Delta; \Gamma \vdash e_0 . m[\vec{\tau}](\vec{e}) : \tau^r}$

[T-FUNCTION] $\frac{p; \Delta \vdash \vec{\tau} \text{ ok} \quad p; \Delta; \Gamma \vdash \vec{e} : \vec{\tau}' \quad \text{mostspecific}_{p;\Delta}(\text{applicable}_{p;\Delta}(m[\vec{\tau}](\vec{\tau}'), \{(md, \text{Object}) \mid md \in p\}))) = m[\overrightarrow{\alpha \text{ extends } N}](\overrightarrow{-:\vec{\tau}^r})}{p; \Delta; \Gamma \vdash m[\vec{\tau}](\vec{e}) : \tau^r}$

Figure A.16: Static Semantics of Core Fortress with Overloading (II)

Subtyping: $p; \Delta \vdash \tau <: \tau$

[S-OBJ] $p; \Delta \vdash \tau <: \text{Object}$

[S-REFL] $p; \Delta \vdash \tau <: \tau$

[S-TRANS]
$$\frac{p; \Delta \vdash \tau_1 <: \tau_2 \quad p; \Delta \vdash \tau_2 <: \tau_3}{p; \Delta \vdash \tau_1 <: \tau_3}$$

[S-VAR] $p; \Delta \vdash \alpha <: \Delta(\alpha)$

[S-TAPP]
$$\frac{- C[\overline{\alpha \text{ extends } -}] - \text{ extends } \{\overrightarrow{N}\} - \in p}{p; \Delta \vdash C[\overrightarrow{\tau}] <: [\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i}$$

Well-formed types: $p; \Delta \vdash \tau \text{ ok}$

[W-OBJ] $p; \Delta \vdash \text{Object ok}$

[W-VAR]
$$\frac{\alpha \in \text{dom}(\Delta)}{p; \Delta \vdash \alpha \text{ ok}}$$

[W-TAPP]
$$\frac{- C[\overline{\alpha \text{ extends } \overrightarrow{N}}] - \in p \quad p; \Delta \vdash \overrightarrow{\tau} \text{ ok} \quad p; \Delta \vdash \overrightarrow{\tau} <: [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{N}}{p; \Delta \vdash C[\overrightarrow{\tau}] \text{ ok}}$$

Most specific definitions: $\text{mostspecific}_{p;\Delta}(\{(md, \tau)\}) = md$

$$\frac{\overrightarrow{md} = m[\overline{(\alpha \text{ extends } \overrightarrow{N})}_1]((\overline{-:\tau^a})_1):\tau_1^r - \cdots m[\overline{(\alpha \text{ extends } \overrightarrow{N})}_n]((\overline{-:\tau^a})_n):\tau_n^r -}{1 \leq i \leq n \quad (\overrightarrow{\tau^a})_i = \tau_{i1}^a \cdots \tau_{im}^a \quad \tau_{i0}^a = \tau_i \quad \forall 0 \leq j \leq m. p; \Delta \vdash \text{meet}(\{\tau_{1j}^a, \dots, \tau_{nj}^a\}, \tau_{ij}^a)}{\text{mostspecific}_{p;\Delta}(\{(md, \tau)\}) = md_i}$$

Applicable definitions: $\text{applicable}_{p;\Delta}(m\overrightarrow{\tau}, \{(md, \tau)\}) = \{(md, \tau)\}$

$$\text{applicable}_{p;\Delta}(m[\overrightarrow{\tau'}](\overrightarrow{\tau''}), S) = \left\{ \begin{array}{l} \{([\overrightarrow{\tau'}/\overrightarrow{\alpha}]md, \tau) \mid \text{ where } (md, \tau) \in S, \\ \quad md = m[\overline{\alpha \text{ extends } \overrightarrow{N}}](x:\overrightarrow{\tau''}): -, \\ \quad p; \Delta \vdash \overrightarrow{\tau''} <: [\overrightarrow{\tau'}/\overrightarrow{\alpha}]\overrightarrow{\tau''} \\ \quad p; \Delta \vdash \overrightarrow{\tau'} <: [\overrightarrow{\tau'}/\overrightarrow{\alpha}]\overrightarrow{N} \end{array} \right.$$

Figure A.17: Static Semantics of Core Fortress with Overloading (III)

Method definition lookup: $\boxed{visible_p / defined_p(C[\vec{\tau}]) = \{\overline{(md, C[\vec{\tau}])}\}}$

$$visible_p(C[\vec{\tau}]) = defined_p(C[\vec{\tau}]) \cup \bigcup_{C'[\vec{\tau}'] \in \{\vec{N}\}} visible_p([\vec{\tau}/\vec{\alpha}]C'[\vec{\tau}']) \quad \text{where } C[\vec{\alpha}] \text{ extends } \{\vec{N}\} \in p$$

$$defined_p(C[\vec{\tau}]) = \{\overline{([\vec{\tau}/\vec{\alpha}]md, C[\vec{\tau}])}\} \quad \text{where } C[\vec{\alpha}] \text{ } \overline{md} \in p$$

Most specific type: $\boxed{p; \Delta \vdash meet(\{\vec{\tau}\}, \tau)}$

$$[MEET] \quad \frac{\tau' \in \{\vec{\tau}\} \quad \forall 1 \leq i \leq |\vec{\tau}|. p; \Delta \vdash \tau' <: \tau_i}{p; \Delta \vdash meet(\{\vec{\tau}\}, \tau)}$$

Bound of type: $\boxed{bound_{\Delta}(\tau) = \sigma}$

$$bound_{\Delta}(\alpha) = \Delta(\alpha)$$

$$bound_{\Delta}(\sigma) = \sigma$$

Figure A.18: Static Semantics of Core Fortress with Overloading (IV)

A.4 Acyclic Core Fortress with Field Definitions

In this section, we define a Fortress core calculus with acyclic type hierarchy and field definitions inside object definitions. We call this calculus *Acyclic Core Fortress with Field Definitions*. Acyclic Core Fortress with Field Definitions is an extension of Basic Core Fortress with acyclic type hierarchy and field definitions inside object definitions.

A.4.1 Syntax

The syntax for Acyclic Core Fortress with Field Definitions is provided in Figure A.19.

A.4.2 Dynamic Semantics

A dynamic semantics for Acyclic Core Fortress with Field Definitions is provided in Figure A.20.

A.4.3 Static Semantics

A static semantics for Acyclic Core Fortress with Field Definitions is provided in Figures A.21, A.22, and A.23.

We proved the type soundness of Acyclic Core Fortress with Field Definitions and the acyclic type hierarchy of a well-type program in Acyclic Core Fortress with Field Definitions using the standard technique of proving a progress theorem and a subject reduction theorem.

α, β		type variables
m		method name
x, y, z		field names
T		trait name
O		object name
τ, τ', τ''	$::= \alpha$	type
	σ	
σ	$::= N$	type that is not a type variable
	$O[\vec{\tau}]$	
N, M, L	$::= T[\vec{\tau}]$	type that can be a type bound
	Object	
e	$::= x$	expression
	self	
	$O[\vec{\tau}](\vec{e})$	
	$e.x$	
	$e.m[\vec{\tau}](\vec{e})$	
md	$::= m[\overrightarrow{\alpha \text{ extends } N}](\overrightarrow{x:\tau}) : \tau = e$	method definition
vd	$::= x:\tau = e$	field definition
td	$::= \text{trait } T[\overrightarrow{\alpha \text{ extends } N}] \text{ extends } \{\overrightarrow{N}\} \overrightarrow{md} \text{ end}$	trait definition
od	$::= \text{object } O[\overrightarrow{\alpha \text{ extends } N}](\overrightarrow{x:\tau}) \text{ extends } \{\overrightarrow{N}\} \overrightarrow{md} \text{ end}$	object definition
d	$::= td$	definition
	od	
p	$::= \overrightarrow{d} e$	program

Figure A.19: Syntax of Acyclic Core Fortress with Field Definitions

Values, intermediate expressions, evaluation contexts, redexes, and trait and object names

v	$::=$	$O[\vec{\tau}]\{x \mapsto \vec{v};\}$	value
ϵ	$::=$	x	intermediate expression
		self	
		$O[\vec{\tau}](\vec{\epsilon})$	
		$\epsilon.x$	
		$\epsilon.m[\vec{\tau}](\vec{\epsilon})$	
		$O[\vec{\tau}]\{x \mapsto \vec{\epsilon}; x \mapsto \vec{\epsilon}\}$	
E	$::=$	\square	evaluation context
		$O[\vec{\tau}](\vec{\epsilon} E \vec{\epsilon})$	
		$E.x$	
		$E.m[\vec{\tau}](\vec{\epsilon})$	
		$\epsilon.m[\vec{\tau}](\vec{\epsilon} E \vec{\epsilon})$	
		$O[\vec{\tau}]\{x \mapsto \vec{v}; x \mapsto E \vec{x} \mapsto \vec{e}\}$	
R	$::=$	$O[\vec{\tau}](\vec{v})$	redex
		$O[\vec{\tau}]\{x \mapsto \vec{v}; x \mapsto v \vec{x} \mapsto \vec{e}\}$	
		$v.x$	
		$v.m[\vec{\tau}](\vec{v})$	
C	$::=$	T	trait name
		O	object name

Evaluation rules: $\boxed{p \vdash E[R] \longrightarrow E[\epsilon]}$

[R-OBJECT]	$\frac{\text{object } O[\overline{\alpha \text{ extends } _}](\overline{x: _}) _ \overline{x': _} = \overline{e'} _ \text{end} \in p}{p \vdash E[O[\vec{\tau}](\vec{v})] \longrightarrow E[O[\vec{\tau}]\{x \mapsto \vec{v}; x' \mapsto [\vec{v}'/\vec{x}][\vec{\tau}'/\vec{\alpha}]\overline{e'}\}]}$
[R-SUB]	$p \vdash E[O[\vec{\tau}]\{x \mapsto \vec{v}; y \mapsto v \vec{z} \mapsto \vec{e}\}] \longrightarrow E[O[\vec{\tau}]\{x \mapsto \vec{v} y \mapsto v; \vec{z} \mapsto [v/y]\overline{e}\}]$
[R-FIELD]	$p \vdash E[O[\vec{\tau}]\{x \mapsto \vec{v};\}.x_i] \longrightarrow E[v_i]$
[R-METHOD]	$\frac{\text{mbody}_p(m[\vec{\tau}'], O[\vec{\tau}]) = \{(x') \rightarrow e\}}{p \vdash E[O[\vec{\tau}]\{x \mapsto \vec{v};\}.m[\vec{\tau}'](\vec{v}')] \longrightarrow E[[\vec{v}'/\vec{x}][O[\vec{\tau}]\{x \mapsto \vec{v};\}/\text{self}][\vec{v}'/\vec{x}']e]}$

Method body lookup: $\boxed{\text{mbody}_p(m[\vec{\tau}'], \tau) = \{(x) \rightarrow e\}}$

[MB-SELF]	$\frac{_ C[\overline{\alpha \text{ extends } _}] _ \overline{md} _ \in p \quad m[\overline{\alpha' \text{ extends } _}](\overline{x': _}): _ = e \in \{\overline{md}\}}{\text{mbody}_p(m[\vec{\tau}'], C[\vec{\tau}]) = \{[\vec{\tau}'/\vec{\alpha}][\vec{\tau}'/\vec{\alpha}](x') \rightarrow e\}}$
[MB-SUPER]	$\frac{_ C[\overline{\alpha \text{ extends } _}] _ \text{extends } \{\vec{K}\} _ \overline{md} _ \in p \quad m \notin \{\overline{\text{Name}}(\overline{md})\}}{\text{mbody}_p(m[\vec{\tau}'], C[\vec{\tau}]) = \bigcup_{N_i \in \{\vec{N}\}} \text{mbody}_p(m[\vec{\tau}'], [\vec{\tau}'/\vec{\alpha}]N_i)}$
[MB-OBJ]	$\text{mbody}_p(m[\vec{\tau}'], \text{Object}) = \emptyset$

Function/method name lookup: $\boxed{\text{Name}(md) = m}$

$\text{Name}(m[\overline{\alpha \text{ extends } \vec{N}}](\overline{x: \vec{\tau}}): \tau = e) = m$

Figure A.20: Dynamic Semantics of Acyclic Core Fortress with Field Definitions

Environments

$$\begin{array}{ll} \Delta ::= \overline{\alpha <: N} & \text{bound environment} \\ \Gamma ::= \overline{x : \tau} & \text{type environment} \end{array}$$

Program typing: $\boxed{\vdash p : \tau}$

$$[\text{T-PROGRAM}] \quad \frac{p = \vec{d} e \quad p; \emptyset; \emptyset \vdash \vec{d} \text{ ok} \quad p; \emptyset; \emptyset \vdash e : \tau \quad \text{acyclic}(\vec{d})}{\vdash p : \tau}$$

Acyclic type hierarchy: $\boxed{\text{acyclic}(\vec{d})}$

$$[\text{ACYCLIC}] \quad \frac{\begin{array}{l} \text{trait } T[\overline{\alpha \text{ extends } N}] \text{ extends } \{\vec{M}\} \text{ end } \in \vec{d} \quad 1 \leq i \leq |\vec{\alpha}| \\ (1) \ M_i \neq T[-] \text{ implies } p; _ \vdash [\vec{\tau}/\vec{\alpha}] M_i \not\prec: T[\vec{\tau}] \\ \quad \text{(The type names excluding self-extensions form an acyclic hierarchy.)} \\ (2) \ M_i = T[-] \text{ implies } M_j \neq T[-] \quad 1 \leq j \leq |\vec{M}| \quad i \neq j \\ (3) \ M_i = T[\vec{\tau}] \text{ implies } (\tau_i = \alpha_i) \vee (\tau_i = N_i) \vee (\tau_i = \alpha_j \wedge \alpha_i = N_j) \quad 1 \leq j \leq |\vec{\alpha}| \quad i \neq j \end{array}}{\text{acyclic}(\vec{d})}$$

Definition typing: $\boxed{p; \emptyset; \emptyset \vdash d \text{ ok}}$

$$[\text{T-TRAITDEF}] \quad \frac{\Delta = \overline{\alpha <: N} \quad p; \Delta \vdash \vec{N} \text{ ok} \quad p; \Delta \vdash \vec{M} \text{ ok} \quad p; \Delta; \text{self} : T[\vec{\alpha}] \vdash T \text{ ok} \vec{m} \vec{d}}{p \vdash \text{oneOwner}(T)} \quad \frac{}{p; \emptyset; \emptyset \vdash \text{trait } T[\overline{\alpha \text{ extends } N}] \text{ extends } \{\vec{M}\} \vec{m} \vec{d} \text{ end ok}}$$

$$[\text{T-OBJECTDEF}] \quad \frac{\begin{array}{l} \Delta = \overline{\alpha <: N} \quad p; \Delta \vdash \vec{N} \text{ ok} \quad p; \Delta \vdash \vec{\tau} \text{ ok} \quad p; \Delta \vdash \vec{M} \text{ ok} \\ \vec{v} \vec{d} = \overline{x' : \tau' = e} \quad p; \Delta; \overline{x : \tau} \ x'_1 : \tau'_1 \dots x'_{i-1} : \tau'_{i-1} \vdash x'_i : \tau'_i = e_i \text{ ok} \quad 1 \leq i \leq n \\ p; \Delta; \text{self} : O[\vec{\alpha}] \ \overline{x : \tau} \ x' : \tau' \vdash O \text{ ok} \vec{m} \vec{d} \quad p \vdash \text{oneOwner}(O) \end{array}}{p; \emptyset; \emptyset \vdash \text{object } O[\overline{\alpha \text{ extends } N}](\overline{x : \tau}) \text{ extends } \{\vec{M}\} \vec{m} \vec{d} \text{ end ok}}$$

One owner for all the visible methods: $\boxed{p \vdash \text{oneOwner}(C)}$

$$[\text{ONEOWNER}] \quad \frac{\forall m \in \text{visible}_p(C). f \text{ only occurs once in } \text{visible}_p(C)}{p \vdash \text{oneOwner}(C)}$$

Method typing: $\boxed{p; \Delta; \Gamma \vdash C \text{ ok} \vec{m} \vec{d}}$

$$[\text{T-METHODDEF}] \quad \frac{\begin{array}{l} _ C[\overline{\alpha' \text{ extends } _}] _ \text{ extends } \{\vec{K}\} _ \in p \quad p \vdash \text{override}(m, \{\vec{M}\}, [\overline{\alpha \text{ extends } N}] \vec{\tau} \rightarrow \tau_0) \\ \Delta' = \Delta \ \overline{\alpha <: N} \quad p; \Delta' \vdash \vec{N} \text{ ok} \quad p; \Delta' \vdash \vec{\tau} \text{ ok} \quad p; \Delta' \vdash \tau_0 \text{ ok} \\ p; \Delta'; \Gamma \ \overline{x : \tau} \vdash e : \tau' \quad p; \Delta' \vdash \tau' <: \tau_0 \end{array}}{p; \Delta; \Gamma \vdash C \text{ ok} \vec{m} [\overline{\alpha \text{ extends } N}](\overline{x : \tau}) : \tau_0 = e}$$

Figure A.21: Static Semantics of Acyclic Core Fortress with Field Definitions (I)

Field typing: $p; \Delta; \Gamma \vdash vd \text{ ok}$

$$\text{[T-FIELDDDEF]} \quad \frac{p; \Delta \vdash \tau \text{ ok} \quad p; \Delta; \Gamma \vdash e : \tau' \quad p; \Delta \vdash \tau' <: \tau}{p; \Delta; \Gamma \vdash x : \tau = e \text{ ok}}$$

Expression typing: $p; \Delta; \Gamma \vdash \epsilon : \tau$

$$\text{[T-VAR]} \quad p; \Delta; \Gamma \vdash x : \Gamma(x)$$

$$\text{[T-SELF]} \quad p; \Delta; \Gamma \vdash \text{self} : \Gamma(\text{self})$$

$$\text{[T-OBJECT]} \quad \frac{\text{object } O[\overrightarrow{\alpha \text{ extends } _}] (\overrightarrow{_} : \overrightarrow{\tau}) \text{ -- end} \in p \quad p; \Delta \vdash O[\overrightarrow{\tau}] \text{ ok}}{p; \Delta; \Gamma \vdash \overrightarrow{\epsilon} : \overrightarrow{\tau} \quad p; \Delta \vdash \overrightarrow{\tau} <: [\overrightarrow{\tau} / \overrightarrow{\alpha}] \overrightarrow{\tau}} \quad \frac{}{p; \Delta; \Gamma \vdash O[\overrightarrow{\tau}](\overrightarrow{\epsilon}) : O[\overrightarrow{\tau}]}$$

$$\text{[T-INT-OBJECT]} \quad \frac{\text{object } O[\overrightarrow{\alpha \text{ extends } _}] (x' : \overrightarrow{\tau}') \text{ -- } x'' : \overrightarrow{\tau}'' = _ \text{ -- end} \in p}{x^1 x^2 = x' x'' \quad \epsilon^1 \epsilon^2 = \epsilon' \epsilon'' \quad p; \Delta; \Gamma \vdash O[\overrightarrow{\tau}](\epsilon') : O[\overrightarrow{\tau}]} \quad \frac{p; \Delta; \Gamma \overrightarrow{x'} : \overrightarrow{\tau'} \quad \overrightarrow{x''} : \overrightarrow{\tau}'' \vdash \overrightarrow{\epsilon} : \overrightarrow{\tau} \quad p; \Delta \vdash [\overrightarrow{\tau} / \overrightarrow{\alpha}] \overrightarrow{\tau} <: [\overrightarrow{\tau} / \overrightarrow{\alpha}] \overrightarrow{\tau}}{p; \Delta; \Gamma \vdash O[\overrightarrow{\tau}]\{x^1 \mapsto \epsilon^1, x^2 \mapsto \epsilon^2\} : O[\overrightarrow{\tau}]}$$

$$\text{[T-FIELD]} \quad \frac{\text{object } O[\overrightarrow{\alpha \text{ extends } _}] (x' : \overrightarrow{\tau}') \text{ -- } x'' : \overrightarrow{\tau}'' = _ \text{ -- end} \in p \quad \overrightarrow{x} = \overrightarrow{x'} \overrightarrow{x''} \quad \overrightarrow{\tau} = \overrightarrow{\tau'} \overrightarrow{\tau}''}{p; \Delta; \Gamma \vdash \epsilon . x_i : [\overrightarrow{\tau} / \overrightarrow{\alpha}] \tau_i} \quad \frac{p; \Delta; \Gamma \vdash \epsilon : \tau_0 \quad \text{bound}_{\Delta}(\tau_0) = O[\overrightarrow{\tau}^0]}{p; \Delta; \Gamma \vdash \epsilon : \tau_0 \quad \text{mtype}_{p; \Delta}(m, \text{bound}_{\Delta}(\tau_0)) = \{\overrightarrow{\alpha \text{ extends } \overrightarrow{N}} \overrightarrow{\tau} \rightarrow \tau_0\}}$$

$$\text{[T-METHOD]} \quad \frac{p; \Delta \vdash \overrightarrow{\tau} \text{ ok} \quad p; \Delta \vdash \overrightarrow{\tau} <: [\overrightarrow{\tau} / \overrightarrow{\alpha}] \overrightarrow{N}}{p; \Delta; \Gamma \vdash \overrightarrow{\epsilon} : \overrightarrow{\tau} \quad p; \Delta \vdash \overrightarrow{\tau} <: [\overrightarrow{\tau} / \overrightarrow{\alpha}] \overrightarrow{\tau}} \quad \frac{}{p; \Delta; \Gamma \vdash \epsilon . m[\overrightarrow{\tau}](\overrightarrow{\epsilon}) : [\overrightarrow{\tau} / \overrightarrow{\alpha}] \tau_0'}$$

Subtyping: $p; \Delta \vdash \tau <: \tau$

$$\text{[S-OBJ]} \quad p; \Delta \vdash \tau <: \text{Object}$$

$$\text{[S-REFL]} \quad p; \Delta \vdash \tau <: \tau$$

$$\text{[S-TRANS]} \quad \frac{p; \Delta \vdash \tau_1 <: \tau_2 \quad p; \Delta \vdash \tau_2 <: \tau_3}{p; \Delta \vdash \tau_1 <: \tau_3}$$

$$\text{[S-VAR]} \quad p; \Delta \vdash \alpha <: \Delta(\alpha)$$

$$\text{[S-TAPP]} \quad \frac{_ \text{ } C[\overrightarrow{\alpha \text{ extends } _}] \text{ -- extends } \{\overrightarrow{N}\} \text{ --} \in p}{p; \Delta \vdash C[\overrightarrow{\tau}] <: [\overrightarrow{\tau} / \overrightarrow{\alpha}] N_i}$$

Figure A.22: Static Semantics of Acyclic Core Fortress with Field Definitions (II)

Well-formed types: $p; \Delta \vdash \tau \text{ ok}$

[W-OBJ] $p; \Delta \vdash \text{Object} \text{ ok}$

[W-VAR] $\frac{\alpha \in \text{dom}(\Delta)}{p; \Delta \vdash \alpha \text{ ok}}$

[W-TAPP] $\frac{- C[\overline{\alpha \text{ extends } \vec{N}}] _ \in p \quad p; \Delta \vdash \vec{\tau} \text{ ok} \quad p; \Delta \vdash \vec{\tau} <: [\vec{\tau}/\vec{\alpha}]\vec{N}}{p; \Delta \vdash C[\vec{\tau}] \text{ ok}}$

Method overriding: $p \vdash \text{override}(m, \{\vec{N}\}, [\overline{\alpha \text{ extends } \vec{N}}] \vec{\tau} \rightarrow \tau)$

[OVERRIDE] $\frac{\bigcup_{L_i \in \{\vec{L}\}} \text{mtype}_{p; \Delta}(m, L_i) = \{[\overline{\beta \text{ extends } \vec{M}}] \vec{\tau}' \rightarrow \tau'_0\} \quad \vec{N} = [\vec{\alpha}/\vec{\beta}]\vec{M} \quad \vec{\tau} = [\vec{\alpha}/\vec{\beta}]\vec{\tau}' \quad p; \Delta \alpha <: \{\vec{N}\} \vdash \tau_0 <: [\vec{\alpha}/\vec{\beta}]\tau'_0}{p \vdash \text{override}(m, \{\vec{L}\}, [\overline{\alpha \text{ extends } \vec{N}}] \vec{\tau} \rightarrow \tau_0)}$

Method type lookup: $\text{mtype}_{p; \Delta}(m, \tau) = \{[\overline{\alpha \text{ extends } \vec{N}}] \vec{\tau} \rightarrow \tau\}$

[MT-SELF] $\frac{- C[\overline{\alpha \text{ extends } _}] _ \overrightarrow{md} _ \in p \quad m[\overline{\beta \text{ extends } \vec{M}}] (_ : \vec{\tau}) : \tau'_0 = _ \in \{\overrightarrow{md}\}}{\text{mtype}_{p; \Delta}(m, C[\vec{\tau}]) = \{[\vec{\tau}/\vec{\alpha}][\overline{\beta \text{ extends } \vec{M}}] \vec{\tau}' \rightarrow \tau'_0\}}$

[MT-SUPER] $\frac{- C[\overline{\alpha \text{ extends } _}] _ \text{extends } \{\vec{K}\} _ \overrightarrow{md} _ \in p \quad m \notin \{\overrightarrow{Name}(md)\}}{\text{mtype}_{p; \Delta}(m, C[\vec{\tau}]) = \bigcup_{N_i \in \{\vec{N}\}} \text{mtype}_{p; \Delta}(m, [\vec{\tau}/\vec{\alpha}]N_i)}$

[MT-OBJ] $\text{mtype}_{p; \Delta}(m, \text{Object}) = \emptyset$

Auxiliary functions for methods: $\text{defined}_p / \text{inherited}_p / \text{visible}_p(C) = \{\overrightarrow{m}\}$

$\text{defined}_p(C) = \{\overrightarrow{Name}(md)\}$ where $- C _ \overrightarrow{md} _ \in p$

$\text{inherited}_p(C) = \biguplus_{N_i \in \{\vec{N}\}} \{m_i \mid m_i \in \text{visible}_p(N_i), m_i \notin \text{defined}_p(C)\}$ where $- C[\overline{\alpha \text{ extends } _}] _ \text{extends } \{\vec{N}\} _ \in p$

$\text{visible}_p(C) = \text{defined}_p(C) \uplus \text{inherited}_p(C)$

Bound of type: $\text{bound}_\Delta(\tau) = \tau$

$\text{bound}_\Delta(\alpha) = \Delta(\alpha)$
 $\text{bound}_\Delta(N) = N$
 $\text{bound}_\Delta(O[\vec{\tau}]) = O[\vec{\tau}]$

Figure A.23: Static Semantics of Acyclic Core Fortress with Field Definitions (III)

Appendix B

Overloaded Functional Declarations

As mentioned in Chapter 33, this appendix proves that the overloading rules discussed in Chapter 33 guarantee no undefined nor ambiguous call at run time.

B.1 Proof of Coercion Resolution for Functions

This section proves that the overloading rules discussed in the previous sections guarantee the static resolution of coercion (described in Section 17.5) is well defined for functions (the case for methods is analogous).

Consider a static function call $f(A)$ at some program point Z and its corresponding dynamic function call $f(X)$. Let Σ be the set of parameter types of function declarations of f that are visible at Z and applicable to the static call $f(A)$. Let Σ' be the set of parameter types of function declarations of f that are visible at Z and applicable with coercion to the static call $f(A)$. Moreover, let σ' be the subset of Σ' for which no type in Σ' is more specific:

$$\sigma' = \{S \in \Sigma' \mid \neg \exists S' \in \Sigma' : S' \triangleleft S\}.$$

We prove the following:

$$|\Sigma| = 0 \text{ and } |\Sigma'| \neq 0 \text{ imply } |\sigma'| = 1.$$

Informally, if no declaration is applicable to a static call but there is a declaration that is applicable with coercion then there exists a single most specific declaration that is applicable with coercion to the static call.

Lemma 1. *Given an acyclic, irreflexive binary relation R on a set S , and a finite nonempty subset A of S , the set $\{a \in A \mid \neg \exists a' \in A : (a', a) \in R\}$ is nonempty.*

Proof. Consider the relation R on S as a directed acyclic graph. Let A represent a subgraph. Then the Lemma amounts to proving that there exists a node in the graph represented by A with no edges pointing to it. This follows from the fact that A is finite and the graph is acyclic. \square

Lemma 2. *If $|\Sigma'| \geq 1$ then $|\sigma'| \geq 1$.*

Proof. Follows from Lemma 1 where S is the set of all types, A is Σ' , and the relation \triangleleft is acyclic and irreflexive. \square

Lemma 3. *If $|\Sigma| = 0$ then $|\sigma'| \leq 1$.*

Proof. For the purpose of contradiction suppose there are two declarations $f(P)$ and $f(Q)$ in σ' . Since both $f(P)$ and $f(Q)$ are applicable with coercion to $f(A)$ and $|\Sigma| = 0$ there must exist a coercion from some type P' to P and a coercion from some type Q' to Q such that $A \preceq P' \cap Q'$. Therefore it is not the case that $P \blacklozenge Q$. By the overloading rules, $P \neq Q$ and either $P \triangleleft Q$ or $Q \triangleleft P$ or for all $P' \in \mathcal{S}$ and $Q' \in \mathcal{T}$ either $P' \blacklozenge Q'$, or there is a declaration $f(P' \cap Q')$ visible at Z . If $P \triangleleft Q$ or $Q \triangleleft P$ then we contradict our assumption. Otherwise, if there exists a declaration $f(P' \cap Q')$ visible at Z then this declaration is applicable to $f(A)$ without coercion. This contradicts $|\Sigma| = 0$. If such a declaration does not exist then it must be the case that $P' \blacklozenge Q'$. Then both $f(P)$ and $f(Q)$ can not be applicable to the call $f(A)$ which is a contradiction. \square

Theorem 1. *If $|\Sigma| = 0$ and $|\Sigma'| \neq 0$ then $|\sigma'| = 1$.*

Proof. Follows from Lemmas 2 and 3. \square

B.2 Proof of Overloading Resolution for Functions

This section proves that the overloading rules for function declarations are sufficient to guarantee no undefined nor ambiguous call at run time (the case for methods is analogous).

Consider a static function call $f(A)$ at some program point Z and its corresponding dynamic function call $f(X)$. Let Δ be the set of parameter types of function declarations of f that are visible at Z and applicable to the dynamic call $f(X)$. Let Σ be the set of parameter types of function declarations of f that are visible at Z and applicable to the static call $f(A)$. Moreover, let σ be the subset of Σ for which no type in Σ is more specific and let δ be the subset of Δ for which no type in Δ is more specific:

$$\begin{aligned} \sigma &= \{S \in \Sigma \mid \neg \exists S' \in \Sigma : S' \prec S\} \\ \delta &= \{D \in \Delta \mid \neg \exists D' \in \Delta : D' \prec D\}. \end{aligned}$$

Below we prove:

$$\begin{aligned} |\Sigma| \neq 0 &\text{ implies } |\sigma| = 1, \text{ and} \\ |\Sigma| \neq 0 &\text{ implies } |\delta| = 1. \end{aligned}$$

Informally, if any declaration is applicable to a static call then there exists a single most specific declaration that is applicable to the static call and a single most specific declaration that is applicable to the corresponding dynamic call.

Lemma 4. $\Sigma \subseteq \Delta$.

Proof. Notice that $X \preceq A$ by type soundness. If $f(P)$ is applicable to the call $f(A)$ then $A \preceq P$. Notice that $X \preceq A$ implies $X \preceq P$. Therefore $f(P)$ is applicable to the call $f(X)$. \square

Lemma 5. *If $|\Delta| \geq 1$ then $|\delta| \geq 1$. Also, if $|\Sigma| \geq 1$ then $|\sigma| \geq 1$.*

Proof. Follows from Lemma 1 where S is the set of all types, A is Δ and Σ respectively, and the relation \prec is acyclic and irreflexive. \square

Lemma 6. *If $|\Sigma| \geq 1$ then $|\delta| \geq 1$.*

Proof. Follows from Lemmas 4 and 5. \square

Lemma 7. $|\sigma| \leq 1$. Also, $|\delta| \leq 1$.

Proof. We prove this for δ , but the case for σ is identical. For the purpose of contradiction suppose there are two declarations $f(P)$ and $f(Q)$ in δ . Since both $f(P)$ and $f(Q)$ are applicable without coercion to the call $f(X)$ we have $X \preceq P \cap Q$. Therefore it is not the case that $P \blacklozenge Q$. By the overloading rules, $P \neq Q$ and either $P \blacklozenge Q$ or there is a declaration $f(P \cap Q)$ visible at Z . Since it cannot be the case that $P \blacklozenge Q$ there must exist a declaration $f(P \cap Q)$ visible at Z . Since $P \cap Q \preceq P$ and $P \cap Q \preceq Q$ we know $f(P \cap Q)$ is applicable without coercion to the call $f(X)$. Since $P \neq Q$ either $P \cap Q \prec P$ or $P \cap Q \prec Q$. Either case contradicts our assumption. \square

Theorem 2. *If $|\Sigma| \neq 0$ then $|\sigma| = 1$. Also, if $|\Sigma| \neq 0$ then $|\delta| = 1$.*

Proof. Follows from Lemmas 5, 6 and 7. \square

Theorem 3. *If $\sigma = \{S\}$ and $\delta = \{D\}$ then $D \preceq S$.*

Proof. If the declaration with parameter type S and the declaration with parameter type D satisfy the Subtype Rule then the theorem is proved. Otherwise, by the definition of σ we have $\sigma \subseteq \Sigma$. Therefore $S \in \Sigma$. By Lemma 4, $S \in \Delta$. Notice that $S, D \in \Delta$ implies $X \preceq S$ and $X \preceq D$. Therefore, $S \not\blacklozenge D$. By the More Specific Rule for Functions, there must exist a declaration with parameter type $S \cap D$. Because $X \preceq (S \cap D)$, $(S \cap D) \in \Delta$. Notice $(S \cap D) \preceq S$ and $(S \cap D) \preceq D$. By the definition of δ , we have $\neg \exists D' \in \Delta : D' \prec D$. In particular, $(S \cap D) \not\prec D$. Therefore $(S \cap D) = D$. \square

Appendix C

Components and APIs

As mentioned in Chapter 22, we formally specify key functionality of the Fortress component system, and illustrate how we can reason about the correctness of the system.

Components One important restriction on components is that no API may be both imported and exported by the same component. Formally, we introduce two functions on components, *imp* and *exp*, that return the imported and exported APIs of the component, respectively. For any component c , $\text{imp}(c) \cap \text{exp}(c) = \emptyset$. This restriction is required throughout to ground the semantics of operations on components, as discussed in Section 22.7.

APIs Other than its identity, the only relevant characteristic of an API a is the set of APIs that it uses, denoted by $\text{uses}(a)$. Because an API a might expose types defined in $\text{uses}(a)$, we require that a component that exports a also exports all APIs in $\text{uses}(a)$ that it does not import. Formally, the following condition holds on the exported APIs of a component c :

$$a \in \text{exp}(c) \wedge a' \in \text{uses}(a) \implies a' \in \text{imp}(c) \cup \text{exp}(c)$$

Link Given a set $C = \{c_1, \dots, c_k\}$ of components, we define a partial function $\text{link}(C)$ that returns the component resulting from c_1 through c_k . If $c = \text{link}(C)$, then $\text{exp}(c) = \bigcup_{c' \in C} \text{exp}(c')$ and $\text{imp}(c) = \bigcup_{c' \in C} \text{imp}(c') - \text{exp}(c)$.

The function *link* is partial because we do not allow arbitrary sets of components to be linked. In particular, Two components cannot be linked if they export the same API.¹ This restriction is made for the sake of simplicity; it allows programmers to link a set of components without having to specify explicitly which constituent exporting an API A provides the implementation exported by the linked component, and which constituent connects to the constituents that import A : only one component exports A , so there is only one choice. Although we lose expressiveness with this design, the user interface to link is vastly simplified, and it is rare that including multiple components that export a given API in a set of linked components is even desirable. We discuss how even such rare cases can be supported in Section 22.8.

For a compound component, in addition to the exported and imported APIs, we want to know what its constituents are. So we introduce another function *cns*, which takes a component and returns the set of its constituents. That is, $\text{cns}(\text{link}(C)) = C$. It is an invariant of the system that for any compound component C (i.e., $\text{cns}(c) \neq \emptyset$), any API imported by any of its constituents is either imported by C or exported by one of its constituents (i.e., $\bigcup_{c' \in \text{cns}(c)} \text{imp}(c') \subseteq \text{imp}(c) \cup \bigcup_{c' \in \text{cns}(c)} \text{exp}(c')$). This property is crucial for executing components, as we discuss below. A simple component (i.e., one produced directly by compilation) has no constituents (i.e., $\text{cns}(c) = \emptyset$).

¹There is one exception to this rule: the special API Upgradable, which is used during upgrades discussed below.

Upgrade A predicate $upg?$ takes two components and indicates whether the first can be upgraded with the second; that is, $upg?(c_t, c_r)$ returns true if and only if c_t can be upgraded with c_r . This predicate captures both the constraints imposed by a component's $isValidUpgrade$ function and the conditions that guarantee the well-formedness of the result. That is,

$$\begin{aligned} upg?(c_t, c_r) &\implies c_t.isValidUpgrade(c_r) \\ &\quad \wedge imp(c_r) \subseteq exp(c_t) \cup imp(c_t) \\ &\quad \wedge exp(c_r) \subset exp(c_t) \\ &\quad \wedge \forall c \in cns(c_t). (exp(c) \subseteq exp(c_r) \vee exp(c) \cap exp(c_r) = \emptyset \vee upg?(c, c_r)) \end{aligned}$$

Visible and Provided We introduce two new functions on components: vis , which returns the APIs of a component that have not been hidden; and $prov$, which returns those visible APIs that are exported by some top-level constituent of the component (or all the exported APIs of a simple component); we say these APIs are *provided* by the component. We need to distinguish provided APIs because they can be imported by the top-level constituents of a component, and thus by a replacement component in an upgrade, while other visible APIs cannot be. Thus, for a compound component c , $prov(c) = vis(c) \cap \bigcup_{c' \in cns(c)} exp(c')$. For a simple component c , $prov(c) = vis(c) = exp(c)$.

Constrain If c is a compound component and $A \subset exp(c)$ is a set of APIs such that $a \in exp(c) \wedge a' \in uses(a) \cap A \implies a \in A$, we define $c' = constrain(c, A)$ such that $exp(c') = exp(c) - A$ and for any component c'' , $upg?(c', c'') \iff upg?(c, c'') \wedge exp(c') \not\subseteq exp(c'')$. The imp , vis , $prov$ and cns functions all have the same values for c and c' . The extra condition on the upgrade compatibility simply captures the restriction we mentioned above, that a replacement component should not export every API exported by the target.

Hide If c is a compound component and $A \subset vis(c)$ is a set of APIs such that $exp(c) \not\subseteq A$ and $a \in vis(c) \wedge a' \in uses(a) \cap A \implies a \in A$, we define $c' = hide(c, A)$ such that $vis(c') = vis(c) - A$, $prov(c') = prov(c) - A$, $exp(c') = exp(c) - A$, and for any component c'' , $upg?(c', c'') \iff upg?(c, c'') \wedge exp(c') \not\subseteq exp(c'') \wedge vis(c'') \subseteq vis(c')$. The additional clause in $upg?(c', c'')$ (compared with that of $constrain$) reflects the hiding of the APIs: we can no longer upgrade APIs that are hidden.

Upgrade The interplay between imported, exported, visible and provided APIs introduces subtleties. In particular, the last of the three conditions imposed for well-formedness of upgrades is modified to state that for any constituent that is not subsumed by a replacement component, either it can be upgraded with the replacement, or its *visible* APIs are disjoint from the APIs exported by the replacement (i.e., it is unaffected by the upgrade). To maintain the invariant that no two constituents export the same API, we need another condition, which was implied by the previous condition when no APIs were constrained or hidden: if the replacement subsumes any constituents of the target, then its exported APIs must exactly match the exported APIs of some subset of the constituents of the target. That is, if $upg?(c_t, c_r) \wedge \exists c \in cns(c_t). exp(c) \subseteq exp(c_r)$ then $exp(c_r) = \bigcup_{c \in C} exp(c)$ for some $C \subset cns(c_t)$. In practice, this restriction is rarely a problem; in most cases, a user wishes to upgrade a target with a new version of a single constituent component, where the APIs exported by the old and new versions are either an exact match, or there are new APIs introduced by the new component that have no implementation in the target.

Appendix D

Rendering of Fortress Identifiers

In order to more closely approximate mathematical notation, Fortress mandates standard rendering for various input elements, particularly for numerals and identifiers, as specified in Section 5.17.

In this Appendix, we describe the detailed rules for rendering an identifier.

If an identifier consists of letters and possibly digits, but no underscores or other connecting punctuation, prime marks, or apostrophes, then the rules are fairly simple:

(a) If the identifier consists of two ASCII capital letters that are the same, possibly followed by digits, then a single capital letter is rendered double-struck, followed by full-sized (not subscripted) digits in roman font.

QQ *is rendered as* \mathbb{Q} RR64 *is rendered as* $\mathbb{R}64$
ZZ *is rendered as* \mathbb{Z} ZZ512 *is rendered as* $\mathbb{Z}512$

(b) Otherwise, if the identifier has more than two characters and begins with a capital letter, then it is rendered in roman font. (Such names are typically used as names of types in Fortress. Note that an identifier cannot consist entirely of capital letters, because such a token is considered to be an operator.)

Integer *is rendered as* Integer Matrix *is rendered as* Matrix
TotalOrder *is rendered as* TotalOrder BooleanAlgebra *is rendered as* BooleanAlgebra
Fred17 *is rendered as* Fred17 R2D2 *is rendered as* R2D2

(c) Otherwise, if the identifier consists of one or more letters followed by one or more digits, then the letters are rendered in italic and the digits are rendered as roman subscripts.

a3 *is rendered as* a_3 foo7 *is rendered as* foo_7
M1 *is rendered as* M_1 z128 *is rendered as* z_{128}
 Ω_{13} *is rendered as* Ω_{13} myFavoriteThings1625 *is rendered as* $myFavoriteThings_{1625}$

(d) The following names are always rendered in roman type out of respect for tradition:

sin cos tan cot sec csc
sinh cosh tanh coth sech csch
arcsin arccos arctan arccot arcsec arccsc
arsinh arcosh artanh arcoth arsech arcsch
arg deg det exp inf sup
lg ln log ged max min

(e) Otherwise the identifier is simply rendered entirely in italic type.

a	is rendered as	<i>a</i>	foobar	is rendered as	<i>foobar</i>
length	is rendered as	<i>length</i>	isInstanceOf	is rendered as	<i>isInstanceOf</i>
foo7a	is rendered as	<i>foo7a</i>	l33tsp33k	is rendered as	<i>l33tsp33k</i>

If the identifier begins or ends with an underscore, or both, but has no other underscores or other connecting punctuation, or prime marks, or apostrophes:

(f) If the identifier, ignoring its underscores, consists of two ASCII capital letters that are the same, possibly followed by one or more digits, then a single capital letter is rendered in sans-serif (for a leading underscore), script (for a trailing underscore), or italic sans-serif (for both a leading and a trailing underscore), and any digits are rendered as roman subscripts.

(g) Otherwise, the identifier without its underscores is rendered in boldface (for a leading underscore), roman (for a trailing underscore), or bold italic (for both a leading and a trailing underscore); except that if the identifier, ignoring its underscores, consists of one or more letters followed by one or more digits, then the digits are rendered as roman subscripts regardless of the underscores.

m_	is rendered as	<i>m</i>	s_	is rendered as	<i>s</i>
km_	is rendered as	<i>km</i>	kg_	is rendered as	<i>kg</i>
V_	is rendered as	<i>V</i>	kW_	is rendered as	<i>kW</i>
_v	is rendered as	<i>v</i>	_foo13	is rendered as	<i>foo</i> ₁₃

(Roman identifiers are typically used for names of SI dimensional units. See sections 6.1.1 and 6.2.1 of [26] for style questions with respect to dimensions and units.)

These last two rules are actually special cases of the following general rules that apply whenever an identifier contains at least one underscore, other connecting punctuation, prime mark, or apostrophe:

An identifier containing underscores is divided into portions by its underscores; in addition, any apostrophe, prime, or double prime character separates portions and is also itself a portion.

(h) If any portion is empty other than the first or last, then the entire identifier is rendered in italics, underscores and all.

Otherwise, the portions are rendered as follows. The idea is that there is a *principal portion* that may be preceded and/or followed by modifiers, and there may also be a *face portion*:

- If the first portion is not empty, *script*, *fraktur*, *sansserif*, or *monospace*, then the principal portion is the first portion and there is no face portion.
- If the first portion is *script*, *fraktur*, *sansserif*, or *monospace*, then the principal portion is the second portion and the face portion is the first portion.
- If the first portion is empty and the second portion is not *script*, *fraktur*, *sansserif*, or *monospace*, then the principal portion is the second portion and there is no face portion.
- Otherwise the principal portion is the third portion and the face portion is the second portion.

If there is no face portion, the principal portion will be rendered in ordinary italics. However, if the first portion is empty (that is, the identifier begins with a leading underscore), then the principal portion is to be rendered in roman boldface. If the last portion is empty (that is, the identifier ends with a trailing underscore), then the principal portion will be roman rather than italic, or bold italic rather than bold.

If there is a face portion, then that describes an alternate typeface to be used in rendering the principal portion. If there is no face portion, but the principal portion consists of two copies of the same letter, then it is rendered as a single letter in a double-struck face (also known as “blackboard bold”), sans-serif, script, or italic sans-serif font according to whether the first and last portions are (not empty, not empty), (empty, not empty), (not empty, empty), or (empty, empty), respectively. Otherwise, if the first portion is empty (that is, the identifier begins with a leading underscore), then the principal portion is to be rendered in a bold version of the selected face, and if the last portion is empty (that

is, the identifier ends with a trailing underscore), then the principal portion to be rendered in an italic (or bold italic) version of the selected face. The bold and italic modifiers may be used only in combination with certain faces; the following are the allowed combinations:

```
script
bold script
fraktur
bold fraktur
double-struck
sans-serif
bold sans-serif
italic sans-serif
bold italic sans-serif
monospace
```

If a combination can be properly rendered, then the principal portion is rendered but not any preceding portions or underscores. If a combination cannot be properly rendered, then the principal portion and all portions and underscores preceding it are rendered all in italics if possible, and otherwise all in some other default face.

If the principal portion consists of a sequence of letters followed by a sequence of digits, then the letters are rendered in the chosen face and the digits are rendered as roman subscripts. Otherwise the entire principal portion is rendered in the chosen face. The remaining portions (excepting the last, if it is empty) are then processed according to the following rules:

- If a portion is `bar`, then a bar is rendered above what has already been rendered, excluding superscripts and subscripts. For example, `x_bar` is rendered as \bar{x} , `x17_bar` is rendered as \bar{x}_{17} , `x_bar_bar` is rendered as $\bar{\bar{x}}$, and `foo_bar` is rendered as \overline{foo} . (Contrast this last with `foo_baz`, which is rendered as $\overline{foo.baz}$.)
- If a portion is `vec`, then a right-pointing arrow is rendered above what has already been rendered, excluding superscripts and subscripts. For example, `v_vec` is rendered as \vec{v} , `v17_vec` is rendered as \vec{v}_{17} , and `zoom_vec` is rendered as \overrightarrow{zoom} .
- If a portion is `hat`, then a hat is rendered above what has already been rendered, excluding superscripts and subscripts. For example, `x17_hat` is rendered as \hat{x}_{17} .
- If a portion is `dot`, then a dot is rendered above what has already been rendered, excluding superscripts and subscripts; but if the preceding portion was also `dot`, then the new dot is rendered appropriately relative to the previous dot(s). Up to four dots will be rendered side-by-side rather than vertically. For example, `a_dot` is rendered as \dot{a} , `a_dot_dot` is rendered as \ddot{a} , `a_dot_dot_dot` is rendered as $\overset{\cdot}{\underset{\cdot}{\underset{\cdot}{a}}}$, `a_dot_dot_dot_dot` is rendered as $\overset{\cdot}{\underset{\cdot}{\underset{\cdot}{\underset{\cdot}{a}}}}$. Also, `a_vec_dot` is rendered as $\vec{\dot{a}}$.
- If a portion is `star`, then an asterisk `*` is rendered as a superscript. For example, `a_star` is rendered as a^* , `a_star_star` is rendered as a^{**} , `ZZ_star` is rendered as \mathbb{Z}^* .
- If a portion is `splat`, then a number sign `#` is rendered as a superscript. For example, `QQ_splat` is rendered as $\mathbb{Q}^\#$.
- If a portion is `prime`, then a prime mark is rendered as a superscript.
- A prime character is treated the same as `prime`, and a double prime character is treated the same as two consecutive `prime` portions. An apostrophe is treated the same as a prime character, but only if all characters following it in the identifier, if any, are also apostrophes. For example, `a'` is rendered as a' , `a13'` is rendered as a'_{13} , and `a''` is rendered as a'' , but `don't` is rendered as $don't$.
- If a portion is `super` and another portion follows, then that other portion is rendered as a superscript in roman type, and enclosed in parentheses if it is all digits.

- If a portion is `sub` and another portion follows, then that other portion is rendered as a subscript in roman type, and enclosed in parentheses if it is all digits, and preceded by a subscript-separating comma if this portion was immediately preceded by another portion that was rendered as a subscript.
- If a portion consists entirely of capital letters and would, if considered by itself as an identifier, be the name of a non-letter Unicode character that would be subject to replacement by preprocessing, then that Unicode character is rendered as a subscript. For example, `id_OPLUS` is rendered as id_{\oplus} , `ZZ_GT` is rendered as $\mathbb{Z}_{>}$, and `QQ_star_LE` is rendered as \mathbb{Q}_{\leq}^* .
- If the portion is the last portion, and the principal portion was a single letter (or two letters indicating a double-struck letter), and none of the preceding rules in this list applies, it is rendered as a subscript in roman type. For example, `T_min` is rendered as T_{\min} . Note that `T_MAX` is rendered simply as `T_MAX`—because all its letters are capital letters, it is considered to be an operator—but `T_sub_MAX` is rendered as T_{MAX} .
- Otherwise, this portion and all succeeding portions are rendered in italics, along with any underscores that appear adjacent to any of them.

Examples:

<code>M</code>	<i>is rendered as</i>	M	<code>_M</code>	<i>is rendered as</i>	\mathbf{M}
<code>v_vec</code>	<i>is rendered as</i>	\vec{v}	<code>_v_vec</code>	<i>is rendered as</i>	$\vec{\mathbf{v}}$
<code>v1</code>	<i>is rendered as</i>	v_1	<code>v_x</code>	<i>is rendered as</i>	v_x
<code>_v1</code>	<i>is rendered as</i>	\mathbf{v}_1	<code>_v_x</code>	<i>is rendered as</i>	\mathbf{v}_x
<code>a_dot</code>	<i>is rendered as</i>	\dot{a}	<code>a_dot_dot</code>	<i>is rendered as</i>	\ddot{a}
<code>a_dot_dot_dot</code>	<i>is rendered as</i>	$\ddot{\ddot{a}}$	<code>a_dot_dot_dot_dot</code>	<i>is rendered as</i>	$\ddot{\ddot{\ddot{a}}}$
<code>a_dot_dot_dot_dot_dot</code>	<i>is rendered as</i>	$\ddot{\ddot{\ddot{\ddot{a}}}}$	<code>p13'</code>	<i>is rendered as</i>	p'_{13}
<code>p'</code>	<i>is rendered as</i>	p'	<code>p_prime</code>	<i>is rendered as</i>	p'
<code>T_min</code>	<i>is rendered as</i>	T_{\min}	<code>T_max</code>	<i>is rendered as</i>	T_{\max}
<code>foo_bar</code>	<i>is rendered as</i>	\overline{foo}	<code>foo_baz</code>	<i>is rendered as</i>	$\overline{foo_baz}$

In this way, through the use of underscore characters and annotation portions delimited by underscores, the programmer can exercise considerable typographical control over the rendering of variable names; but if no underscores are used, the rendering rules are quite simple.

Appendix E

Support for Unicode Input in ASCII

To facilitate the writing of Fortress programs using ASCII-based tools, Fortress programs are subject to *ASCII conversion*, an idempotent process consisting of three steps described in this appendix. We expect that IDEs that support Unicode may apply ASCII conversion immediately, saving programs as converted Unicode character sequences.

E.1 Word Pasting across Line Breaks

Consider any line terminator in the program for which the last non-whitespace character before the line terminator is an ampersand (&) that is immediately preceded by a word character, and the first non-whitespace character after the line terminator is an ampersand that is immediately followed by a word character. All the characters between the ampersands inclusive are removed from the program. Note that the removed characters other than the two ampersands must be whitespace characters (including one or more line terminators). This step permits very long names and numerals to be split across line boundaries. For example:

```
supercalifragilisticexpiali&
  &docious = 0.142857142857142857&
  &142857 TIMES &
    GREEK_SMALL_LETTER_&

  &UPSILON_WITH_DIALYTICA_AND_TONOS
```

becomes

```
supercalifragilisticexpialidocious = 0.142857142857142857142857 TIMES &
  GREEK_SMALL_LETTER_UPSILON_WITH_DIALYTICA_AND_TONOS
```

To ensure the idempotence of ASCII conversion, it is a static error if an ampersand is immediately followed by another ampersand, or if it is immediately followed by one or more whitespace characters, not including a line terminator, followed by an ASCII letter or ampersand. In other words, if an ampersand is not the last non-whitespace character on a line then the next non-whitespace character must not be an ampersand, nor can it be an ASCII letter unless it is adjacent to the ampersand.

E.2 Converting Names of Unicode Characters and ASCII Shorthand

After word pasting across line breaks, certain ASCII character sequences are converted into single Unicode characters. These character sequences fall into two categories: restricted words (or parts of restricted words) and ASCII shorthand for operators and special characters. In some cases, ampersands adjacent to converted sequences are removed.

E.2.1 Determining Potentially Convertible Character Sequences

We first identify nonoverlapping contiguous subsequences of the program that may be subject to conversion in this step. In particular, names of characters and ASCII shorthand consist of only ASCII characters, and they are *not* converted within string literals.

The boundaries of string literals (and comments) are determined follows: There are three modes of processing: outside any comment or string literal, inside a string literal and inside a comment. Within a comment, we also keep track of “nesting depth” (this is 0 when not within a comment). Processing proceeds from left to right. Outside any comment or string literal, encountering an unescaped string literal delimiter (i.e., a string literal delimiter not immediately preceded by an unescaped backslash) changes processing to the mode for within a string literal (however, it is a static error if the string literal delimiter is the right double quotation mark), and encountering the opening comment delimiter “`(*)`” changes processing to the mode for within a comment, incrementing the nesting depth (to 1). All other characters are ignored, except to note they are not within a string literal. In particular, because character literal delimiters are ignored, string literal delimiters must be escaped within character literals.

Within a string literal, all characters other than an unescaped string literal delimiter are ignored (except to note that they are within a string literal). An unescaped string literal delimiter switches processing to the mode for outside any comment or string literal. However, it is a static error if a line terminator appears within a string literal (Fortress provides *escape sequences* to include line terminators in strings; see Section 5.10), or if the opening and closing string literal delimiters do not “match” (again see Section 5.10).

Within a comment, all characters, including unescaped string delimiters, are ignored (except to note that they are not within a string literal) other than the two-character opening and closing comment delimiters “`(*)`” and “`*)`”. Whenever the opening comment delimiter is encountered, the nesting depth is incremented, and each time the closing comment delimiter is encountered, the nesting depth is decremented, until it becomes 0. At that point, processing is changed again to the mode for outside any comment or string literal.

A contiguous subsequence of ASCII characters not within any string literal is a candidate for conversion in this step if it is either a restricted word, or it is a maximal such subsequence that does not contain any restricted-word characters, control characters, whitespace characters or ampersands (i.e., it must be delimited by whitespace, a control or non-ASCII characters, string literals, restricted words, ampersands, or the beginning or end of the file). We call the latter kind of subsequence an *ASCII symbol sequence*. The conversion of restricted words and ASCII symbol sequences are described in Section E.2.2 and Section E.2.3 respectively. Whitespace characters and ampersands are not converted, but some ampersands may be eliminated, as described in Section E.2.4.

E.2.2 Converting Restricted Words

A restricted word that “names” a Unicode character that is not protected¹ (i.e., not an ASCII or control character nor a string literal delimiter) is replaced by that character. A restricted word may name a character in various ways, described below. In a few cases, a restricted word may name more than one character (via different ways), so the order in which we consider these ways is important. Also, in some cases, only part of the restricted word is replaced by a Unicode character.

¹Protecting the backslash (an ASCII character) and string literal delimiters preserves the boundaries of string literals determined as described above; protecting the other ASCII characters ensures that the ASCII conversion process is idempotent.

Fortress explicitly provides short ASCII names for many characters, especially ones that programmers might most commonly want. Some characters have more than one short name. For operators, these names are given in Appendix F, including the following common ones:

LE	becomes	\leq	GE	becomes	\geq	NE	becomes	\neq
BY	becomes	\times	TIMES	becomes	\times	CROSS	becomes	\times
DOT	becomes	\cdot	PRODUCT	becomes	\prod	SUM	becomes	\sum
CUP	becomes	\cup	CAP	becomes	\cap	SUBSET	becomes	\subset
EMPTYSET	becomes	\emptyset	AND	becomes	\wedge	OR	becomes	\vee

To provide a level of compatibility with Fortran, Appendix F also gives short names for the following ASCII (and thus protected) characters:

LT	becomes	$<$	GT	becomes	$>$	EQ	becomes	$=$
----	---------	-----	----	---------	-----	----	---------	-----

These names are the only ones converted to a protected character. Furthermore, they are *not* replaced by the corresponding character unless they are delimited by whitespace characters (*not* ampersands). Thus, they cannot participate in further conversion, maintaining the idempotence of ASCII conversion.

The following non-operator characters also have short names:

ALPHA	becomes	A	alpha	becomes	α	BOTTOM	becomes	\perp
BETA	becomes	B	beta	becomes	β	TOP	becomes	\top
GAMMA	becomes	Γ	gamma	becomes	γ	INF	becomes	∞
DELTA	becomes	Δ	delta	becomes	δ	FORALL	becomes	\forall
EPSILON	becomes	E	epsilon	becomes	ϵ	EXISTS	becomes	\exists
ZETA	becomes	Z	zeta	becomes	ζ			
ETA	becomes	H	eta	becomes	η			
THETA	becomes	Θ	theta	becomes	θ			
IOTA	becomes	I	iota	becomes	ι			
KAPPA	becomes	K	kappa	becomes	κ			
LAMBDA	becomes	Λ	lambda	becomes	λ			
MU	becomes	M	mu	becomes	μ			
NU	becomes	N	nu	becomes	ν			
XI	becomes	Ξ	xi	becomes	ξ			
OMICRON	becomes	O	omicron	becomes	o			
PI	becomes	Π	pi	becomes	π			
RHO	becomes	P	rho	becomes	ρ			
SIGMA	becomes	Σ	sigma	becomes	σ			
TAU	becomes	T	tau	becomes	τ			
UPSILON	becomes	Υ	upsilon	becomes	υ			
PHI	becomes	Φ	phi	becomes	ϕ			
CHI	becomes	X	chi	becomes	χ			
PSI	becomes	Ψ	psi	becomes	ψ			
OMEGA	becomes	Ω	omega	becomes	ω			

A restricted word that is not a short name specified by Fortress is compared to names of characters specified by the Unicode Standard [27]. Because restricted words consist only of letters, digits and underscores, while Unicode names may include hyphens and spaces (but not underscores), hyphens and spaces in the Unicode names are replaced by underscores for this comparison. Specifically, if a restricted word is the official Unicode 5.0 name of a character with hyphens and spaces replaced by underscores, it is replaced by that character. For any Unicode character other than the control characters, there is a unique official Unicode 5.0 name not shared by any other Unicode character. Since control characters are protected, they do not present a problem in this regard.

If there is no matching official Unicode 5.0 name, the restricted word is compared to the alternative names for characters specified by the Unicode Standard, again with hyphens and spaces replaced by underscores, and converted to a

character with such a matching name. If more than one character has a matching alternative name, then the restricted word is converted to the first such non-protected character (i.e., the one with the smallest code point). If there is no matching alternative name, then the restricted word is compared to official Unicode 5.0 names and any alternative names specified in the Unicode Standard, with underscores in place of hyphens and spaces, where any of the following substrings may be omitted:

```
"LETTER_"
"DIGIT_"
"RADICAL_"
"NUMERAL_"
```

If there are multiple such substrings in a given name, any combination of them may be omitted. Again, if there are multiple characters with matching abridged names, the restricted word is converted to the first such non-protected character.

If the restricted word is not converted to a single Unicode character by any of the above rules, parts of it may be converted as follows: If the restricted word begins with the short name (i.e., the name in the table above) of a Greek letter followed by an underscore or a digit, or ends with the short name of a Greek letter that is preceded by an underscore, or contains the short name of a Greek letter with an underscore on each side of it, then the short name of the Greek letter is replaced by the Greek letter itself. In the same manner, the character sequence “micro” is replaced with the character MICRO SIGN μ (U+00B5, which looks just like the Greek lowercase mu μ but is different). If a word-part being thus replaced has an underscore to each side, and the underscore on the right is the last character of the restricted word, then the underscore on the left is removed as the name is replaced; this ad hoc rule is for the sake of the abbreviations of certain dimensional units, so that, for example, `micro.OMEGA_` is converted to $\mu\Omega_$, signifying micro-ohms, and `G_OMEGA_` is converted to $G\Omega_$, signifying gigaohms. Here are some other examples:

<code>alpha</code>	<i>becomes</i>	α	<code>OMEGA13</code>	<i>becomes</i>	Ω_{13}
<code>alpha_hat</code>	<i>becomes</i>	α_{hat}	<code>theta_elephant</code>	<i>becomes</i>	$\theta_{elephant}$
<code>OMEGA_</code>	<i>becomes</i>	$\Omega_$	<code>_XI</code>	<i>becomes</i>	Ξ

E.2.3 Converting ASCII Symbol Sequences

For the sequences of characters other than restricted words, each is converted from left to right, with the longest possible substring being converted at once, with one exception: The sequence “(<” is not converted if it is immediately followed by any of the following characters: ‘<’, ‘|’, ‘/’, ‘\’, ‘*’, or ‘.’. That is, with this one exception, the longest shorthand beginning from the first character, if any, is converted first. Then, the longest shorthand beginning from the next character of the string after replacement is converted, and so on. Here are the ASCII shorthands for some of the characters we expect to be most frequently used:

<code>[\</code>	<i>becomes</i>	<code>[[</code>	<code>\]</code>	<i>becomes</i>	<code>]]</code>
<code>-></code>	<i>becomes</i>	<code>→</code>	<code>=></code>	<i>becomes</i>	<code>⇒</code>
<code>~></code>	<i>becomes</i>	<code>↔</code>	<code> -></code>	<i>becomes</i>	<code>↦</code>
<code>>=</code>	<i>becomes</i>	<code>≥</code>	<code><=</code>	<i>becomes</i>	<code>≤</code>
<code>≠</code>	<i>becomes</i>	<code>≠</code>	<code>...</code>	<i>becomes</i>	<code>...</code>
<code>:=</code>	<i>becomes</i>	<code>:=</code>			

Although the characters with string literals are generally not subject to this step of ASCII conversion, they are if they are part of a restricted-word escape sequence or a quoted-character escape sequence. See Section 5.10 for details.

E.2.4 Eliminating Ampersands

Finally, if an ampersand is adjacent to a sequence of characters that is changed by this step of ASCII conversion (even if the sequence was only partly changed, as long as the character adjacent to the ampersand is changed), the

ampersand is removed after the transformation, *unless* the ampersand is the first or last non-whitespace character on the line. Thus, the expression

```
(GREEK_SMALL_LETTER_PHI&GREEK_SMALL_LETTER_PSI +  
GREEK_SMALL_LETTER_OMEGA&GREEK_SMALL_LETTER_LAMBDA)
```

is converted to:

```
(φψ + ωλ)
```

in which there are two identifiers, each consisting of two Greek letters. On the other hand,

```
(GREEK_SMALL_LETTER_PHI GREEK_SMALL_LETTER_PSI +  
GREEK_SMALL_LETTER_OMEGA GREEK_SMALL_LETTER_LAMBDA)
```

is converted to:

```
(φ ψ + ω λ)
```

in which there are four identifiers.

Because we never convert to protected characters, and converted sequences consist only of protected characters, this step is idempotent: repeated application has no effect beyond the first.

E.3 Apostrophe Replacement within Numerals

The last step of ASCII conversion simply replaces apostrophes with digit-group separators when they appear in words that would otherwise be numerals. Specifically, consider any word that is not within a string literal (determined as described in Section E.2.1), has only digits, letters, underscores and apostrophes, and neither begins nor ends with an apostrophe or an underscore. Call such a word a *proto-numeral*. If a proto-numeral begins with a digit or ends with a radix specifier (see Section 5.13), or if it is immediately followed by a '.' and then a proto-numeral that ends with a radix specifier, or it is immediately preceded by a '.', which is immediately preceded by a proto-numeral that begins with a digit, then replace each apostrophe in that proto-numeral with a digit-group separator.

E.4 Equivalence under ASCII Conversion

Two (valid) Fortress programs are equivalent under ASCII conversion if they are identical after ASCII conversion, except for string literals (determined as described in Section E.2.1), and the converted programs have string literals in exactly corresponding places, and corresponding string literals evaluate to the same string. This exception and additional rule is necessary because string literals are not subject to ASCII conversion.

Appendix F

Operator Precedence, Chaining, and Enclosure

This appendix contains the detailed rules for which Unicode 5.0 characters may be used as operators, which operators form enclosing pairs, which operators may be chained, and what precedence relationships exist among the various operators. (If no precedence relationship is stated explicitly for any given pair of operators, then there is no precedence relationship between those two operators. Remember that precedence is not transitive in Fortress.)

In each of the character lists below, each line gives the Unicode codepoint, the full Unicode 5.0 name, a rendering of the character in \TeX (if possible), and any ASCII shorthand or short names for the character.

F.1 Bracket Pairs for Enclosing Operators

Here are the bracket pairs that may be used as enclosing operators. Note that there is one group of four brackets; within that group, either left bracket may be paired with either right bracket to form an enclosing operator.

U+005B	LEFT SQUARE BRACKET	[
U+005D	RIGHT SQUARE BRACKET]	
U+007B	LEFT CURLY BRACKET	{	
U+007D	RIGHT CURLY BRACKET	}	
U+2045	LEFT SQUARE BRACKET WITH QUILL	[./	
U+2046	RIGHT SQUARE BRACKET WITH QUILL	/.]	
U+2308	LEFT CEILING	[/
U+2309	RIGHT CEILING]	\
U+230A	LEFT FLOOR	[\
U+230B	RIGHT FLOOR]	/
U+27C5	LEFT S-SHAPED BAG DELIMITER	.	\
U+27C6	RIGHT S-SHAPED BAG DELIMITER	/.	
U+27E8	MATHEMATICAL LEFT ANGLE BRACKET	<	<
U+27E9	MATHEMATICAL RIGHT ANGLE BRACKET	>	>
U+27EA	MATHEMATICAL LEFT DOUBLE ANGLE BRACKET	<<	<<
U+27EB	MATHEMATICAL RIGHT DOUBLE ANGLE BRACKET	>>	>>

U+2983	LEFT WHITE CURLY BRACKET	{	{\
U+2984	RIGHT WHITE CURLY BRACKET	}	\}
U+2985	LEFT WHITE PARENTHESIS		(\
U+2986	RIGHT WHITE PARENTHESIS		\)
U+2987	Z NOTATION LEFT IMAGE BRACKET		(/
U+2988	Z NOTATION RIGHT IMAGE BRACKET		/)
U+2989	Z NOTATION LEFT BINDING BRACKET		<
U+298A	Z NOTATION RIGHT BINDING BRACKET		>
U+298B	LEFT SQUARE BRACKET WITH UNDERBAR		[.\
U+298C	RIGHT SQUARE BRACKET WITH UNDERBAR		\.]
U+298D	LEFT SQUARE BRACKET WITH TICK IN TOP CORNER		[././
U+298E	RIGHT SQUARE BRACKET WITH TICK IN BOTTOM CORNER		/./.]
U+298F	LEFT SQUARE BRACKET WITH TICK IN BOTTOM CORNER		[.\.\
U+2990	RIGHT SQUARE BRACKET WITH TICK IN TOP CORNER		\.\.]
U+2991	LEFT ANGLE BRACKET WITH DOT		<.
U+2992	RIGHT ANGLE BRACKET WITH DOT		. >
U+2993	LEFT ARC LESS-THAN BRACKET		(<
U+2994	RIGHT ARC GREATER-THAN BRACKET		>)
U+2995	DOUBLE LEFT ARC GREATER-THAN BRACKET		((>
U+2996	DOUBLE RIGHT ARC LESS-THAN BRACKET		<))
U+2997	LEFT BLACK TORTOISE SHELL BRACKET		[*/
U+2998	RIGHT BLACK TORTOISE SHELL BRACKET		/*
U+29D8	LEFT WIGGLY FENCE		[/\//
U+29D9	RIGHT WIGGLY FENCE		//\]
U+29DA	LEFT DOUBLE WIGGLY FENCE		[/\//\//
U+29DB	RIGHT DOUBLE WIGGLY FENCE		//\//\]
U+29FC	LEFT-POINTING CURVED ANGLE BRACKET		<
U+29FD	RIGHT-POINTING CURVED ANGLE BRACKET		>
U+300C	LEFT CORNER BRACKET	┌	</
U+300D	RIGHT CORNER BRACKET	└	\>
U+300E	LEFT WHITE CORNER BRACKET		<</
U+300F	RIGHT WHITE CORNER BRACKET		\>>
U+3010	LEFT BLACK LENTICULAR BRACKET		{*/
U+3011	RIGHT BLACK LENTICULAR BRACKET		/*}
U+3018	LEFT WHITE TORTOISE SHELL BRACKET		[//
U+3014	LEFT TORTOISE SHELL BRACKET		[/
U+3015	RIGHT TORTOISE SHELL BRACKET		/]
U+3019	RIGHT WHITE TORTOISE SHELL BRACKET		//]
U+3016	LEFT WHITE LENTICULAR BRACKET		{/
U+3017	RIGHT WHITE LENTICULAR BRACKET		/}

F.2 Vertical-Line Operators

The following are vertical-line operators:

U+007C	VERTICAL LINE		
U+2016	DOUBLE VERTICAL LINE		
U+2AF4	TRIPLE VERTICAL BAR BINARY RELATION		

F.3 Arithmetic Operators

F.3.1 Multiplication and Division

The following are multiplication operators. Note that `ASTERISK OPERATOR` is always a multiplication operator; `ASTERISK` is treated as a synonym for `ASTERISK OPERATOR` where appropriate, but `ASTERISK` also has other uses, for example in the ASCII bracket encodings `[*/ and /*]` and `{*/ and /*}`.

U+002A	ASTERISK	*	
U+00B7	MIDDLE DOT	·	DOT
U+00D7	MULTIPLICATION SIGN	×	TIMES BY
U+2217	ASTERISK OPERATOR	*	
U+228D	MULTISET MULTIPLICATION		
U+2297	CIRCLED TIMES	⊗	OTIMES
U+2299	CIRCLED DOT OPERATOR	⊙	ODOT
U+229B	CIRCLED ASTERISK OPERATOR	⊛	CIRCLEDAST
U+22A0	SQUARED TIMES	⊠	BOXTIMES
U+22A1	SQUARED DOT OPERATOR	⊡	BOXDOT
U+22C5	DOT OPERATOR	·	
U+29C6	SQUARED ASTERISK		BOXAST
U+29D4	TIMES WITH LEFT HALF BLACK		
U+29D5	TIMES WITH RIGHT HALF BLACK		
U+2A2F	VECTOR OR CROSS PRODUCT	×	CROSS
U+2A30	MULTIPLICATION SIGN WITH DOT ABOVE		DOTTIMES
U+2A31	MULTIPLICATION SIGN WITH UNDERBAR		
U+2A34	MULTIPLICATION SIGN IN LEFT HALF CIRCLE		
U+2A35	MULTIPLICATION SIGN IN RIGHT HALF CIRCLE		
U+2A36	CIRCLED MULTIPLICATION SIGN WITH CIRCUMFLEX ACCENT		
U+2A37	MULTIPLICATION SIGN IN DOUBLE CIRCLE		
U+2A3B	MULTIPLICATION SIGN IN TRIANGLE		TRITIMES

The following are division operators. Note that `DIVISION SLASH` is always a division operator; `SOLIDUS` is treated as a synonym for `DIVISION SLASH` where appropriate, but `SOLIDUS` also has other uses, for example in the ASCII bracket encodings `(/ and /)` and `[/ and /]` and `{/ and /}`.

U+002F	SOLIDUS	/	
U+00F7	DIVISION SIGN	÷	DIV
U+2215	DIVISION SLASH	/	
U+2298	CIRCLED DIVISION SLASH	⊘	OSLASH
U+29B8	CIRCLED REVERSE SOLIDUS		
U+29BC	CIRCLED ANTICLOCKWISE-ROTATED DIVISION SIGN		
U+29C4	SQUARED RISING DIAGONAL SLASH		BOXSLASH
U+29F5	REVERSE SOLIDUS OPERATOR	\	
U+29F8	BIG SOLIDUS	/	
U+29F9	BIG REVERSE SOLIDUS	\	
U+2A38	CIRCLED DIVISION SIGN		ODIV

U+2AFD	DOUBLE SOLIDUS OPERATOR	//	//
U+2AFB	TRIPLE SOLIDUS BINARY RELATION		///

Note also that `per` is treated as a division operator.

F.3.2 Addition and Subtraction

The following three operators have the same precedence and may be mixed.

U+002B	PLUS SIGN	+
U+002D	HYPHEN-MINUS	-
U+2212	MINUS SIGN	-

They each have lower precedence than any of the following multiplication and division operators:

U+002A	ASTERISK	*
U+002F	SOLIDUS	/
U+00B7	MIDDLE DOT	· DOT
U+00D7	MULTIPLICATION SIGN	× TIMES
U+00F7	DIVISION SIGN	÷ DIV
U+2215	DIVISION SLASH	/
U+2217	ASTERISK OPERATOR	*
U+22C5	DOT OPERATOR	·
U+2A2F	VECTOR OR CROSS PRODUCT	× CROSS

The following two operators have the same precedence and may be mixed.

U+2214	DOT PLUS	⊕ DOTPLUS
U+2238	DOT MINUS	⊖ DOTMINUS

They each have lower precedence than this multiplication operator:

U+2A30	MULTIPLICATION SIGN WITH DOT ABOVE	⊙ DOTTIMES
--------	------------------------------------	------------

The following two operators have the same precedence and may be mixed.

U+2A25	PLUS SIGN WITH DOT BELOW
U+2A2A	MINUS SIGN WITH DOT BELOW

The following two operators have the same precedence and may be mixed.

U+2A39	PLUS SIGN IN TRIANGLE	⊕ TRIPLUS
U+2A3A	MINUS SIGN IN TRIANGLE	⊖ TRIMINUS

They each have lower precedence than this multiplication operator:

U+2A3B	MULTIPLICATION SIGN IN TRIANGLE	⊙ TRITIMES
--------	---------------------------------	------------

The following two operators have the same precedence and may be mixed.

U+2295	CIRCLED PLUS	⊕ OPLUS
U+2296	CIRCLED MINUS	⊖ OMINUS

They each have lower precedence than any of the following multiplication and division operators:

U+2297	CIRCLED TIMES	⊗ OTIMES
U+2298	CIRCLED DIVISION SLASH	⊘ OSLASH
U+2299	CIRCLED DOT OPERATOR	⊙ ODOT
U+229B	CIRCLED ASTERISK OPERATOR	⊗ CIRCLEDAST
U+2A38	CIRCLED DIVISION SIGN	⊘ ODIV

The following two operators have the same precedence and may be mixed.

U+229E SQUARED PLUS	\boxplus	BOXPLUS
U+229F SQUARED MINUS	\boxminus	BOXMINUS

They each have lower precedence than any of these multiplication or division operators:

U+22A0 SQUARED TIMES	\boxtimes	BOXTIMES
U+22A1 SQUARED DOT OPERATOR	\boxdot	BOXDOT
U+29C4 SQUARED RISING DIAGONAL SLASH	\boxslash	BOXSLASH
U+29C6 SQUARED ASTERISK	\boxast	BOXAST

These are other miscellaneous addition and subtraction operators:

U+00B1 PLUS-MINUS SIGN	\pm
U+2213 MINUS-OR-PLUS SIGN	\mp
U+2242 MINUS TILDE	
U+2A22 PLUS SIGN WITH SMALL CIRCLE ABOVE	$\overset{\circ}{+}$
U+2A23 PLUS SIGN WITH CIRCUMFLEX ACCENT ABOVE	$\overset{\wedge}{+}$
U+2A24 PLUS SIGN WITH TILDE ABOVE	$\overset{\sim}{+}$
U+2A26 PLUS SIGN WITH TILDE BELOW	$\underset{\sim}{+}$
U+2A27 PLUS SIGN WITH SUBSCRIPT TWO	$+_2$
U+2A28 PLUS SIGN WITH BLACK TRIANGLE	
U+2A29 MINUS SIGN WITH COMMA ABOVE	$\overset{,}{-}$
U+2A2B MINUS SIGN WITH FALLING DOTS	
U+2A2C MINUS SIGN WITH RISING DOTS	
U+2A2D PLUS SIGN IN LEFT HALF CIRCLE	
U+2A2E PLUS SIGN IN RIGHT HALF CIRCLE	

F.3.3 Miscellaneous Arithmetic Operators

The operators MAX, MIN, REM, MOD, GCD, LCM, CHOOSE, and `per`, none of which corresponds to a single Unicode character, are considered to be arithmetic operators, having higher precedence than certain relational operators, as described in a later section.

F.3.4 Set Intersection, Union, and Difference

The following are the set intersection operators:

U+2229 INTERSECTION	\cap	CAP INTERSECT
U+22D2 DOUBLE INTERSECTION	\mcap	CAPCAP
U+2A40 INTERSECTION WITH DOT		
U+2A43 INTERSECTION WITH OVERBAR	$\bar{\cap}$	
U+2A44 INTERSECTION WITH LOGICAL AND		
U+2A4B INTERSECTION BESIDE AND JOINED WITH INTERSECTION		
U+2A4D CLOSED INTERSECTION WITH SERIFS		
U+2ADB TRANSVERSAL INTERSECTION		

The following are the set union operators:

U+222A UNION
 U+228E MULTISSET UNION
 U+22D3 DOUBLE UNION
 U+2A41 UNION WITH MINUS SIGN
 U+2A42 UNION WITH OVERBAR
 U+2A45 UNION WITH LOGICAL OR
 U+2A4A UNION BESIDE AND JOINED WITH UNION
 U+2A4C CLOSED UNION WITH SERIFS
 U+2A50 CLOSED UNION WITH SERIFS AND SMASH PRODUCT

∪ CUP UNION
 ⊕ UPLUS
 ∩ CUPCUP
 ⋃

They each have lower precedence than any of the set intersection operators.

This is a miscellaneous set operator:

U+2216 SET MINUS

\ SETMINUS

F.3.5 Square Arithmetic Operators

The following are the square intersection operators:

U+2293 SQUARE CAP
 U+2A4E DOUBLE SQUARE INTERSECTION

⊐ SQCAP
 ⊓ SQCAPCAP

The following are the square union operators:

U+2294 SQUARE CUP
 U+2A4F DOUBLE SQUARE UNION

⊑ SQCUP
 ⊔ SQCUPCUP

They each have lower precedence than either of the square intersection operators.

F.3.6 Curly Arithmetic Operators

The following is the curly intersection operator:

U+22CF CURLY LOGICAL AND

⋈ CURLYAND

The following is the curly union operator:

U+22CE CURLY LOGICAL OR

⋉ CURLYOR

It has lower precedence than the curly intersection operator.

F.4 Relational Operators

F.4.1 Equivalence and Inequivalence Operators

Every operator listed in this section has lower precedence than any operator listed in Section F.3.

The following are equivalence operators. They may be chained. Moreover, they may be chained with any other single group of chainable relational operators, as described in later sections.

U+003D	EQUALS SIGN	=	EQ
U+2243	ASYMPTOTICALLY EQUAL TO	≈	SIMEQ
U+2245	APPROXIMATELY EQUAL TO	≈	
U+2246	APPROXIMATELY BUT NOT ACTUALLY EQUAL TO	≈	
U+2247	NEITHER APPROXIMATELY NOR ACTUALLY EQUAL TO	≉	
U+2248	ALMOST EQUAL TO	≈	APPROX
U+224A	ALMOST EQUAL OR EQUAL TO	≈	APPROXEQ
U+224C	ALL EQUAL TO		
U+224D	EQUIVALENT TO	×	
U+224E	GEOMETRICALLY EQUIVALENT TO	≈	BUMPEQV
U+2251	GEOMETRICALLY EQUAL TO	∴	DOTEQDOT
U+2252	APPROXIMATELY EQUAL TO OR THE IMAGE OF	∴	
U+2253	IMAGE OF OR APPROXIMATELY EQUAL TO	∴	
U+2256	RING IN EQUAL TO	≡	EQRING
U+2257	RING EQUAL TO	≡	RINGEQ
U+225B	STAR EQUALS		
U+225C	DELTA EQUAL TO	△	EQDEL
U+225D	EQUAL TO BY DEFINITION		EQDEF
U+225F	QUESTIONED EQUAL TO		
U+2261	IDENTICAL TO	≡	EQV EQUIV
U+2263	STRICTLY EQUIVALENT TO	≡	SEQV
U+229C	CIRCLED EQUALS		
U+22CD	REVERSED TILDE EQUALS	≅	
U+22D5	EQUAL AND PARALLEL TO		
U+29E3	EQUALS SIGN AND SLANTED PARALLEL		
U+29E4	EQUALS SIGN AND SLANTED PARALLEL WITH TILDE ABOVE		
U+29E5	IDENTICAL TO AND SLANTED PARALLEL		
U+2A66	EQUALS SIGN WITH DOT BELOW		
U+2A67	IDENTICAL WITH DOT ABOVE		
U+2A6C	SIMILAR MINUS SIMILAR		
U+2A6E	EQUALS WITH ASTERISK		
U+2A6F	ALMOST EQUAL TO WITH CIRCUMFLEX ACCENT		
U+2A70	APPROXIMATELY EQUAL OR EQUAL TO		
U+2A71	EQUALS SIGN ABOVE PLUS SIGN		
U+2A72	PLUS SIGN ABOVE EQUALS SIGN		
U+2A73	EQUALS SIGN ABOVE TILDE OPERATOR		
U+2A75	TWO CONSECUTIVE EQUALS SIGNS		
U+2A76	THREE CONSECUTIVE EQUALS SIGNS		
U+2A77	EQUALS SIGN WITH TWO DOTS ABOVE AND TWO DOTS BELOW		
U+2A78	EQUIVALENT WITH FOUR DOTS ABOVE		
U+2AAE	EQUALS SIGN WITH BUMPY ABOVE		
U+FE66	SMALL EQUALS SIGN		
U+FF1D	FULLWIDTH EQUALS SIGN		

The following are inequivalence operators. They may not be chained.

U+2244	NOT ASYMPTOTICALLY EQUAL TO	≉	NSIMEQ
U+2249	NOT ALMOST EQUAL TO	≉	NAPPROX
U+2260	NOT EQUAL TO	≠	=/= NE
U+2262	NOT IDENTICAL TO	≠	NEQV
U+226D	NOT EQUIVALENT TO	×	

F.4.2 Plain Comparison Operators

Every operator listed in this section has lower precedence than any operator listed in Sections F.3.1, F.3.2, and F.3.3.

The following are less-than operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+003C LESS-THAN SIGN	<	LT
U+2264 LESS-THAN OR EQUAL TO	≤	LE
U+2266 LESS-THAN OVER EQUAL TO	≧	
U+2268 LESS-THAN BUT NOT EQUAL TO	≨	
U+226A MUCH LESS-THAN	≪	<<
U+2272 LESS-THAN OR EQUIVALENT TO	≦	
U+22D6 LESS-THAN WITH DOT	⋖	DOTLT
U+22D8 VERY MUCH LESS-THAN	≪≪	<<<
U+22DC EQUAL TO OR LESS-THAN		
U+22E6 LESS-THAN BUT NOT EQUIVALENT TO	≧̸	
U+29C0 CIRCLED LESS-THAN	⊲	
U+2A79 LESS-THAN WITH CIRCLE INSIDE		
U+2A7B LESS-THAN WITH QUESTION MARK ABOVE		
U+2A7D LESS-THAN OR SLANTED EQUAL TO		
U+2A7F LESS-THAN OR SLANTED EQUAL TO WITH DOT INSIDE		
U+2A81 LESS-THAN OR SLANTED EQUAL TO WITH DOT ABOVE		
U+2A83 LESS-THAN OR SLANTED EQUAL TO WITH DOT ABOVE RIGHT		
U+2A85 LESS-THAN OR APPROXIMATE		
U+2A87 LESS-THAN AND SINGLE-LINE NOT EQUAL TO		
U+2A89 LESS-THAN AND NOT APPROXIMATE		
U+2A8D LESS-THAN ABOVE SIMILAR OR EQUAL		
U+2A95 SLANTED EQUAL TO OR LESS-THAN		
U+2A97 SLANTED EQUAL TO OR LESS-THAN WITH DOT INSIDE		
U+2A99 DOUBLE-LINE EQUAL TO OR LESS-THAN		
U+2A9B DOUBLE-LINE SLANTED EQUAL TO OR LESS-THAN		
U+2A9D SIMILAR OR LESS-THAN		
U+2A9F SIMILAR ABOVE LESS-THAN ABOVE EQUALS SIGN		
U+2AA1 DOUBLE NESTED LESS-THAN		
U+2AA3 DOUBLE NESTED LESS-THAN WITH UNDERBAR		
U+2AA6 LESS-THAN CLOSED BY CURVE		
U+2AA8 LESS-THAN CLOSED BY CURVE ABOVE SLANTED EQUAL		
U+2AF7 TRIPLE NESTED LESS-THAN		
U+2AF9 DOUBLE-LINE SLANTED LESS-THAN OR EQUAL TO		
U+FE64 SMALL LESS-THAN SIGN		
U+FF1C FULLWIDTH LESS-THAN SIGN		

The following are greater-than operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+003E GREATER-THAN SIGN	>	GT
U+2265 GREATER-THAN OR EQUAL TO	≥	GE
U+2267 GREATER-THAN OVER EQUAL TO	≦	
U+2269 GREATER-THAN BUT NOT EQUAL TO	≧	
U+226B MUCH GREATER-THAN	≫	>>
U+2273 GREATER-THAN OR EQUIVALENT TO	≧	
U+22D7 GREATER-THAN WITH DOT	⋗	DOTGT
U+22D9 VERY MUCH GREATER-THAN	≫≫	>>>

U+22DD	EQUAL TO OR GREATER-THAN	
U+22E7	GREATER-THAN BUT NOT EQUIVALENT TO	ℳ
U+29C1	CIRCLED GREATER-THAN	
U+2A7A	GREATER-THAN WITH CIRCLE INSIDE	
U+2A7C	GREATER-THAN WITH QUESTION MARK ABOVE	
U+2A7E	GREATER-THAN OR SLANTED EQUAL TO	
U+2A80	GREATER-THAN OR SLANTED EQUAL TO WITH DOT INSIDE	
U+2A82	GREATER-THAN OR SLANTED EQUAL TO WITH DOT ABOVE	
U+2A84	GREATER-THAN OR SLANTED EQUAL TO WITH DOT ABOVE LEFT	
U+2A86	GREATER-THAN OR APPROXIMATE	
U+2A88	GREATER-THAN AND SINGLE-LINE NOT EQUAL TO	
U+2A8A	GREATER-THAN AND NOT APPROXIMATE	
U+2A8E	GREATER-THAN ABOVE SIMILAR OR EQUAL	
U+2A96	SLANTED EQUAL TO OR GREATER-THAN	
U+2A98	SLANTED EQUAL TO OR GREATER-THAN WITH DOT INSIDE	
U+2A9A	DOUBLE-LINE EQUAL TO OR GREATER-THAN	
U+2A9C	DOUBLE-LINE SLANTED EQUAL TO OR GREATER-THAN	
U+2A9E	SIMILAR OR GREATER-THAN	
U+2AA0	SIMILAR ABOVE GREATER-THAN ABOVE EQUALS SIGN	
U+2AA2	DOUBLE NESTED GREATER-THAN	
U+2AA7	GREATER-THAN CLOSED BY CURVE	
U+2AA9	GREATER-THAN CLOSED BY CURVE ABOVE SLANTED EQUAL	
U+2AF8	TRIPLE NESTED GREATER-THAN	
U+2AFA	DOUBLE-LINE SLANTED GREATER-THAN OR EQUAL TO	
U+FE65	SMALL GREATER-THAN SIGN	
U+FF1E	FULLWIDTH GREATER-THAN SIGN	

The following are miscellaneous plain comparison operators. They may not be mixed or chained.

U+226E	NOT LESS-THAN	ℵ	NLT
U+226F	NOT GREATER-THAN	ℵ	NGT
U+2270	NEITHER LESS-THAN NOR EQUAL TO	ℵ	NLE
U+2271	NEITHER GREATER-THAN NOR EQUAL TO	ℵ	NGE
U+2274	NEITHER LESS-THAN NOR EQUIVALENT TO	ℵ	
U+2275	NEITHER GREATER-THAN NOR EQUIVALENT TO	ℵ	
U+2276	LESS-THAN OR GREATER-THAN	ℵ	
U+2277	GREATER-THAN OR LESS-THAN	ℵ	
U+2278	NEITHER LESS-THAN NOR GREATER-THAN	ℵ	
U+2279	NEITHER GREATER-THAN NOR LESS-THAN	ℵ	
U+22DA	LESS-THAN EQUAL TO OR GREATER-THAN	ℵ	
U+22DB	GREATER-THAN EQUAL TO OR LESS-THAN	ℵ	
U+2A8B	LESS-THAN ABOVE DOUBLE-LINE EQUAL ABOVE GREATER-THAN	ℵ	
U+2A8C	GREATER-THAN ABOVE DOUBLE-LINE EQUAL ABOVE LESS-THAN	ℵ	
U+2A8F	LESS-THAN ABOVE SIMILAR ABOVE GREATER-THAN	ℵ	
U+2A90	GREATER-THAN ABOVE SIMILAR ABOVE LESS-THAN	ℵ	
U+2A91	LESS-THAN ABOVE GREATER-THAN ABOVE DOUBLE-LINE EQUAL	ℵ	
U+2A92	GREATER-THAN ABOVE LESS-THAN ABOVE DOUBLE-LINE EQUAL	ℵ	
U+2A93	LESS-THAN ABOVE SLANTED EQUAL ABOVE GREATER-THAN ABOVE SLANTED EQUAL	ℵ	
U+2A94	GREATER-THAN ABOVE SLANTED EQUAL ABOVE LESS-THAN ABOVE SLANTED EQUAL	ℵ	
U+2AA4	GREATER-THAN OVERLAPPING LESS-THAN	ℵ	
U+2AA5	GREATER-THAN BESIDE LESS-THAN	ℵ	

The following are not really comparison operators, but it is convenient to list them here because they also have lower

precedence than any operator listed in Sections F.3.1, F.3.2, and F.3.3:

U+0023	NUMBER SIGN	#
U+003A	COLON	:

F.4.3 Set Comparison Operators

Every operator listed in this section has lower precedence than any operator listed in Section F.3.4.

The following are subset comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+2282	SUBSET OF	\subset	SUBSET
U+2286	SUBSET OF OR EQUAL TO	\subseteq	SUBSETEQ
U+228A	SUBSET OF WITH NOT EQUAL TO	\subsetneq	SUBSETNEQ
U+22D0	DOUBLE SUBSET	\Subset	SUBSUB
U+27C3	OPEN SUBSET		
U+2ABD	SUBSET WITH DOT		
U+2ABF	SUBSET WITH PLUS SIGN BELOW		
U+2AC1	SUBSET WITH MULTIPLICATION SIGN BELOW		
U+2AC3	SUBSET OF OR EQUAL TO WITH DOT ABOVE		
U+2AC5	SUBSET OF ABOVE EQUALS SIGN		
U+2AC7	SUBSET OF ABOVE TILDE OPERATOR		
U+2AC9	SUBSET OF ABOVE ALMOST EQUAL TO		
U+2ACB	SUBSET OF ABOVE NOT EQUAL TO		
U+2ACF	CLOSED SUBSET		
U+2AD1	CLOSED SUBSET OR EQUAL TO		
U+2AD5	SUBSET ABOVE SUBSET		

The following are superset comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+2283	SUPERSET OF	\supset	SUPERSET
U+2287	SUPERSET OF OR EQUAL TO	\supseteq	SUPERSETEQ
U+228B	SUPERSET OF WITH NOT EQUAL TO	\supsetneq	SUPERSETNEQ
U+22D1	DOUBLE SUPERSET	\Supset	SUPSUP
U+27C4	OPEN SUPERSET		
U+2ABE	SUPERSET WITH DOT		
U+2AC0	SUPERSET WITH PLUS SIGN BELOW		
U+2AC2	SUPERSET WITH MULTIPLICATION SIGN BELOW		
U+2AC4	SUPERSET OF OR EQUAL TO WITH DOT ABOVE		
U+2AC6	SUPERSET OF ABOVE EQUALS SIGN		
U+2AC8	SUPERSET OF ABOVE TILDE OPERATOR		
U+2ACA	SUPERSET OF ABOVE ALMOST EQUAL TO		
U+2ACC	SUPERSET OF ABOVE NOT EQUAL TO		
U+2AD0	CLOSED SUPERSET		
U+2AD2	CLOSED SUPERSET OR EQUAL TO		
U+2AD6	SUPERSET ABOVE SUPERSET		

The following are miscellaneous set comparison operators. They may not be mixed or chained.

U+2284	NOT A SUBSET OF	$\not\subset$	NSUBSET
U+2285	NOT A SUPERSET OF	$\not\supset$	NSUPSET
U+2288	NEITHER A SUBSET OF NOR EQUAL TO	$\not\subseteq$	NSUBSETEQ

U+2289	NEITHER A SUPERSET OF NOR EQUAL TO	$\not\supseteq$	NSUPSETEQ
U+2AD3	SUBSET ABOVE SUPERSET		
U+2AD4	SUPERSET ABOVE SUBSET		
U+2AD7	SUPERSET BESIDE SUBSET		
U+2AD8	SUPERSET BESIDE AND JOINED BY DASH WITH SUBSET		

F.4.4 Square Comparison Operators

Every operator listed in this section has lower precedence than any operator listed in Section F.3.5.

The following are square “image of” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+228F	SQUARE IMAGE OF	\sqsubset	SQSUBSET
U+2291	SQUARE IMAGE OF OR EQUAL TO	\sqsubseteq	SQSUBSETEQ
U+22E4	SQUARE IMAGE OF OR NOT EQUAL TO		

The following are square “original of” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+2290	SQUARE ORIGINAL OF	\sqsupset	SQSUPSET
U+2292	SQUARE ORIGINAL OF OR EQUAL TO	\sqsupseteq	SQSUPSETEQ
U+22E5	SQUARE ORIGINAL OF OR NOT EQUAL TO		

The following are miscellaneous square comparison operators. They may not be mixed or chained.

U+22E2	NOT SQUARE IMAGE OF OR EQUAL TO	$\not\sqsubseteq$	
U+22E3	NOT SQUARE ORIGINAL OF OR EQUAL TO	$\not\sqsupseteq$	

F.4.5 Curly Comparison Operators

Every operator listed in this section has lower precedence than any operator listed in Section F.3.6.

The following are curly “precedes” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+227A	PRECEDES	\prec	PREC
U+227C	PRECEDES OR EQUAL TO	\preceq	PRECEQ
U+227E	PRECEDES OR EQUIVALENT TO	\precsim	PRECSIM
U+22B0	PRECEDES UNDER RELATION		
U+22DE	EQUAL TO OR PRECEDES	\preceq	EQPREC
U+22E8	PRECEDES BUT NOT EQUIVALENT TO	\precsim	PRECNSIM
U+2AAF	PRECEDES ABOVE SINGLE-LINE EQUALS SIGN		
U+2AB1	PRECEDES ABOVE SINGLE-LINE NOT EQUAL TO		
U+2AB3	PRECEDES ABOVE EQUALS SIGN		
U+2AB5	PRECEDES ABOVE NOT EQUAL TO		
U+2AB7	PRECEDES ABOVE ALMOST EQUAL TO		
U+2AB9	PRECEDES ABOVE NOT ALMOST EQUAL TO		
U+2ABB	DOUBLE PRECEDES		

The following are curly “succeeds” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+227B	SUCCEEDS	\succ	SUCC
U+227D	SUCCEEDS OR EQUAL TO	\succcurlyeq	SUCCEQ
U+227F	SUCCEEDS OR EQUIVALENT TO	\succsim	SUCCSIM
U+22B1	SUCCEEDS UNDER RELATION		
U+22DF	EQUAL TO OR SUCCEEDS	\succcurlyeq	EQSUCC
U+22E9	SUCCEEDS BUT NOT EQUIVALENT TO	$\succ\neq$	SUCCNSIM
U+2AB0	SUCCEEDS ABOVE SINGLE-LINE EQUALS SIGN		
U+2AB2	SUCCEEDS ABOVE SINGLE-LINE NOT EQUAL TO		
U+2AB4	SUCCEEDS ABOVE EQUALS SIGN		
U+2AB6	SUCCEEDS ABOVE NOT EQUAL TO		
U+2AB8	SUCCEEDS ABOVE ALMOST EQUAL TO		
U+2ABA	SUCCEEDS ABOVE NOT ALMOST EQUAL TO		
U+2ABC	DOUBLE SUCCEEDS		

The following are miscellaneous curly comparison operators. They may not be mixed or chained.

U+2280	DOES NOT PRECEDE	\nprec	NPREC
U+2281	DOES NOT SUCCEED	\nprec	NSUCC
U+22E0	DOES NOT PRECEDE OR EQUAL	\nprec	
U+22E1	DOES NOT SUCCEED OR EQUAL	\nprec	

F.4.6 Triangular Comparison Operators

The following are triangular “subgroup” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+22B2	NORMAL SUBGROUP OF	\triangleleft
U+22B4	NORMAL SUBGROUP OF OR EQUAL TO	\trianglelefteq

The following are triangular “contains as subgroup” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+22B3	CONTAINS AS NORMAL SUBGROUP	\triangleright
U+22B5	CONTAINS AS NORMAL SUBGROUP OR EQUAL TO	\trianglerighteq

The following are miscellaneous triangular comparison operators. They may not be mixed or chained.

U+22EA	NOT NORMAL SUBGROUP OF	\ntriangleleft
U+22EB	DOES NOT CONTAIN AS NORMAL SUBGROUP	\ntriangleleft
U+22EC	NOT NORMAL SUBGROUP OF OR EQUAL TO	\ntriangleleft
U+22ED	DOES NOT CONTAIN AS NORMAL SUBGROUP OR EQUAL	\ntriangleleft

F.4.7 Chickenfoot Comparison Operators

The following are chickenfoot “smaller than” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+2AAA	SMALLER THAN	\llcorner	SMALLER
U+2AAC	SMALLER THAN OR EQUAL TO	\llcorner	SMALLEREQ

The following are chickenfoot “larger than” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+2AAB	LARGER THAN	\lrcorner	LARGER
U+2AAD	LARGER THAN OR EQUAL TO	\lrcorner	LARGEREQ

F.4.8 Miscellaneous Relational Operators

The following operators are considered to be relational operators, having higher precedence than certain boolean operators, as described in a later section.

U+2208	ELEMENT OF	\in	IN
U+2209	NOT AN ELEMENT OF	\notin	NOT IN
U+220A	SMALL ELEMENT OF	ϵ	
U+220B	CONTAINS AS MEMBER	\ni	CONTAINS
U+220C	DOES NOT CONTAIN AS MEMBER	$\not\ni$	
U+220D	SMALL CONTAINS AS MEMBER	\ni	
U+22F2	ELEMENT OF WITH LONG HORIZONTAL STROKE		
U+22F3	ELEMENT OF WITH VERTICAL BAR AT END OF HORIZONTAL STROKE		
U+22F4	SMALL ELEMENT OF WITH VERTICAL BAR AT END OF HORIZONTAL STROKE		
U+22F5	ELEMENT OF WITH DOT ABOVE	$\dot{\in}$	
U+22F6	ELEMENT OF WITH OVERBAR	$\overline{\in}$	
U+22F7	SMALL ELEMENT OF WITH OVERBAR	$\overline{\epsilon}$	
U+22F8	ELEMENT OF WITH UNDERBAR	$\underline{\in}$	
U+22F9	ELEMENT OF WITH TWO HORIZONTAL STROKES		
U+22FA	CONTAINS WITH LONG HORIZONTAL STROKE		
U+22FB	CONTAINS WITH VERTICAL BAR AT END OF HORIZONTAL STROKE		
U+22FC	SMALL CONTAINS WITH VERTICAL BAR AT END OF HORIZONTAL STROKE		
U+22FD	CONTAINS WITH OVERBAR	$\overline{\ni}$	
U+22FE	SMALL CONTAINS WITH OVERBAR	$\overline{\ni}$	
U+22FF	Z NOTATION BAG MEMBERSHIP		

F.5 Boolean Operators

Every operator listed in this section has lower precedence than any operator listed in Section F.4.

The following are the Boolean conjunction operators:

U+2227	LOGICAL AND	\wedge	AND
U+27D1	AND WITH DOT		
U+2A51	LOGICAL AND WITH DOT ABOVE	$\dot{\wedge}$	
U+2A53	DOUBLE LOGICAL AND		
U+2A55	TWO INTERSECTING LOGICAL AND	\pitchfork	
U+2A5A	LOGICAL AND WITH MIDDLE STEM		
U+2A5C	LOGICAL AND WITH HORIZONTAL DASH		
U+2A5E	LOGICAL AND WITH DOUBLE OVERBAR		
U+2A60	LOGICAL AND WITH DOUBLE UNDERBAR		

The following are the Boolean disjunction operators:

U+2228	LOGICAL OR	\vee	OR
U+2A52	LOGICAL OR WITH DOT ABOVE	$\dot{\vee}$	
U+2A54	DOUBLE LOGICAL OR		
U+2A56	TWO INTERSECTING LOGICAL OR	\pitchfork	
U+2A5B	LOGICAL OR WITH MIDDLE STEM		
U+2A5D	LOGICAL OR WITH HORIZONTAL DASH		
U+2A62	LOGICAL OR WITH DOUBLE OVERBAR		
U+2A63	LOGICAL OR WITH DOUBLE UNDERBAR		

They each have lower precedence than any of the Boolean conjunction operators.

The following are miscellaneous Boolean operators:

U+2192	RIGHTWARDS ARROW	→	->	IMPLIES
U+2194	LEFT RIGHT ARROW	↔	<->	IFF
U+22BB	XOR	∨		
U+22BC	NAND	⋀		
U+22BD	NOR	∇		

F.6 Other Operators

Each of the following operators has no defined precedence relationships to any of the other operators listed in this appendix.

U+0021	EXCLAMATION MARK	!		
U+0024	DOLLAR SIGN	\$		
U+0025	PERCENT SIGN	%		
U+003F	QUESTION MARK	?		
U+0040	COMMERCIAL AT	@		
U+005E	CIRCUMFLEX ACCENT	^		
U+007E	TILDE	~		
U+00A1	INVERTED EXCLAMATION MARK	¡		
U+00A2	CENT SIGN			CENTS
U+00A3	POUND SIGN			
U+00A4	CURRENCY SIGN			
U+00A5	YEN SIGN			
U+00A6	BROKEN BAR			
U+00AC	NOT SIGN	¬		NOT
U+00B0	DEGREE SIGN	°		DEGREES
U+00BF	INVERTED QUESTION MARK	¿		
U+203C	DOUBLE EXCLAMATION MARK	!!	!!	
U+2190	LEFTWARDS ARROW	←	<-	
U+2191	UPWARDS ARROW	↑	UPARROW	
U+2193	DOWNWARDS ARROW	↓	DOWNARROW	
U+2195	UP DOWN ARROW	↕	UPDOWNARROW	
U+2196	NORTH WEST ARROW	↖	NWARROW	
U+2197	NORTH EAST ARROW	↗	NEARROW	
U+2198	SOUTH EAST ARROW	↘	SEARROW	
U+2199	SOUTH WEST ARROW	↙	SWARROW	
U+219A	LEFTWARDS ARROW WITH STROKE	↔	<-/-	
U+219B	RIGHTWARDS ARROW WITH STROKE	↠	-/->	
U+219C	LEFTWARDS WAVE ARROW			
U+219D	RIGHTWARDS WAVE ARROW			
U+219E	LEFTWARDS TWO HEADED ARROW			
U+219F	UPWARDS TWO HEADED ARROW			
U+21A0	RIGHTWARDS TWO HEADED ARROW			
U+21A1	DOWNWARDS TWO HEADED ARROW			
U+21A2	LEFTWARDS ARROW WITH TAIL			
U+21A3	RIGHTWARDS ARROW WITH TAIL			
U+21A4	LEFTWARDS ARROW FROM BAR	↵	LEADSTO	

U+21A5	UPWARDS ARROW FROM BAR	
U+21A7	DOWNWARDS ARROW FROM BAR	
U+21A8	UP DOWN ARROW WITH BASE	
U+21A9	LEFTWARDS ARROW WITH HOOK	
U+21AA	RIGHTWARDS ARROW WITH HOOK	
U+21AB	LEFTWARDS ARROW WITH LOOP	
U+21AC	RIGHTWARDS ARROW WITH LOOP	
U+21AD	LEFT RIGHT WAVE ARROW	
U+21AE	LEFT RIGHT ARROW WITH STROKE	
U+21AF	DOWNWARDS ZIGZAG ARROW	
U+21B0	UPWARDS ARROW WITH TIP LEFTWARDS	
U+21B1	UPWARDS ARROW WITH TIP RIGHTWARDS	
U+21B2	DOWNWARDS ARROW WITH TIP LEFTWARDS	
U+21B3	DOWNWARDS ARROW WITH TIP RIGHTWARDS	
U+21B4	RIGHTWARDS ARROW WITH CORNER DOWNWARDS	
U+21B5	DOWNWARDS ARROW WITH CORNER LEFTWARDS	
U+21B6	ANTICLOCKWISE TOP SEMICIRCLE ARROW	
U+21B7	CLOCKWISE TOP SEMICIRCLE ARROW	
U+21B8	NORTH WEST ARROW TO LONG BAR	
U+21B9	LEFTWARDS ARROW TO BAR OVER RIGHTWARDS ARROW TO BAR	
U+21BA	ANTICLOCKWISE OPEN CIRCLE ARROW	
U+21BB	CLOCKWISE OPEN CIRCLE ARROW	
U+21BC	LEFTWARDS HARPOON WITH BARB UPWARDS	↵ LEFTHARPOONUP
U+21BD	LEFTWARDS HARPOON WITH BARB DOWNWARDS	↶ LEFTHARPOONDOWN
U+21BE	UPWARDS HARPOON WITH BARB RIGHTWARDS	↗ UPHARPOONRIGHT
U+21BF	UPWARDS HARPOON WITH BARB LEFTWARDS	↖ UPHARPOONLEFT
U+21C0	RIGHTWARDS HARPOON WITH BARB UPWARDS	↘ RIGHTHARPOONUP
U+21C1	RIGHTWARDS HARPOON WITH BARB DOWNWARDS	↙ RIGHTHARPOONDOWN
U+21C2	DOWNWARDS HARPOON WITH BARB RIGHTWARDS	↴ DOWNHARPOONRIGHT
U+21C3	DOWNWARDS HARPOON WITH BARB LEFTWARDS	↳ DOWNHARPOONLEFT
U+21C4	RIGHTWARDS ARROW OVER LEFTWARDS ARROW	⇔ RIGHTLEFTARROWS
U+21C5	UPWARDS ARROW LEFTWARDS OF DOWNWARDS ARROW	
U+21C6	LEFTWARDS ARROW OVER RIGHTWARDS ARROW	↔ LEFTRIGHTARROWS
U+21C7	LEFTWARDS PAIRED ARROWS	⇐ LEFTLEFTARROWS
U+21C8	UPWARDS PAIRED ARROWS	⇑ UPUPARROWS
U+21C9	RIGHTWARDS PAIRED ARROWS	⇒ RIGHTRIGHTARROWS
U+21CA	DOWNWARDS PAIRED ARROWS	⇓ DOWNDOWNARROWS
U+21CB	LEFTWARDS HARPOON OVER RIGHTWARDS HARPOON	
U+21CC	RIGHTWARDS HARPOON OVER LEFTWARDS HARPOON	⇚ RIGHTLEFTHARPOONS
U+21CD	LEFTWARDS DOUBLE ARROW WITH STROKE	⇏
U+21CE	LEFT RIGHT DOUBLE ARROW WITH STROKE	⇎
U+21CF	RIGHTWARDS DOUBLE ARROW WITH STROKE	⇐
U+21D0	LEFTWARDS DOUBLE ARROW	←
U+21D1	UPWARDS DOUBLE ARROW	↑
U+21D2	RIGHTWARDS DOUBLE ARROW	⇒ =>
U+21D3	DOWNWARDS DOUBLE ARROW	↓
U+21D4	LEFT RIGHT DOUBLE ARROW	⇔ <=>
U+21D5	UP DOWN DOUBLE ARROW	⇕
U+21D6	NORTH WEST DOUBLE ARROW	
U+21D7	NORTH EAST DOUBLE ARROW	
U+21D8	SOUTH EAST DOUBLE ARROW	
U+21D9	SOUTH WEST DOUBLE ARROW	

U+21DA	LEFTWARDS TRIPLE ARROW	\Leftarrow
U+21DB	RIGHTWARDS TRIPLE ARROW	\Rightarrow
U+21DC	LEFTWARDS SQUIGGLE ARROW	
U+21DD	RIGHTWARDS SQUIGGLE ARROW	\rightsquigarrow
U+21DE	UPWARDS ARROW WITH DOUBLE STROKE	
U+21DF	DOWNWARDS ARROW WITH DOUBLE STROKE	
U+21E0	LEFTWARDS DASHED ARROW	\dashleftarrow
U+21E1	UPWARDS DASHED ARROW	
U+21E2	RIGHTWARDS DASHED ARROW	\dashrightarrow
U+21E3	DOWNWARDS DASHED ARROW	
U+21E4	LEFTWARDS ARROW TO BAR	
U+21E5	RIGHTWARDS ARROW TO BAR	
U+21E6	LEFTWARDS WHITE ARROW	
U+21E7	UPWARDS WHITE ARROW	
U+21E8	RIGHTWARDS WHITE ARROW	
U+21E9	DOWNWARDS WHITE ARROW	
U+21EA	UPWARDS WHITE ARROW FROM BAR	
U+21EB	UPWARDS WHITE ARROW ON PEDESTAL	
U+21EC	UPWARDS WHITE ARROW ON PEDESTAL WITH HORIZONTAL BAR	
U+21ED	UPWARDS WHITE ARROW ON PEDESTAL WITH VERTICAL BAR	
U+21EE	UPWARDS WHITE DOUBLE ARROW	
U+21EF	UPWARDS WHITE DOUBLE ARROW ON PEDESTAL	
U+21F0	RIGHTWARDS WHITE ARROW FROM WALL	
U+21F1	NORTH WEST ARROW TO CORNER	
U+21F2	SOUTH EAST ARROW TO CORNER	
U+21F3	UP DOWN WHITE ARROW	
U+21F4	RIGHT ARROW WITH SMALL CIRCLE	
U+21F5	DOWNWARDS ARROW LEFTWARDS OF UPWARDS ARROW	
U+21F6	THREE RIGHTWARDS ARROWS	
U+21F7	LEFTWARDS ARROW WITH VERTICAL STROKE	
U+21F8	RIGHTWARDS ARROW WITH VERTICAL STROKE	
U+21F9	LEFT RIGHT ARROW WITH VERTICAL STROKE	
U+21FA	LEFTWARDS ARROW WITH DOUBLE VERTICAL STROKE	
U+21FB	RIGHTWARDS ARROW WITH DOUBLE VERTICAL STROKE	
U+21FC	LEFT RIGHT ARROW WITH DOUBLE VERTICAL STROKE	
U+21FD	LEFTWARDS OPEN-HEADED ARROW	
U+21FE	RIGHTWARDS OPEN-HEADED ARROW	
U+21FF	LEFT RIGHT OPEN-HEADED ARROW	
U+2201	COMPLEMENT	\complement
U+2202	PARTIAL DIFFERENTIAL	∂ DEL
U+2204	THERE DOES NOT EXIST	\nexists
U+2206	INCREMENT	Δ
U+220F	N-ARY PRODUCT	\prod PRODUCT
U+2210	N-ARY COPRODUCT	\coprod COPRODUCT
U+2211	N-ARY SUMMATION	\sum SUM
U+2218	RING OPERATOR	\circ CIRC RING COMPOSE
U+2219	BULLET OPERATOR	\bullet BULLET
U+221A	SQUARE ROOT	$\sqrt{\quad}$ SQRT
U+221B	CUBE ROOT	$\sqrt[3]{\quad}$ CBRT
U+221C	FOURTH ROOT	$\sqrt[4]{\quad}$ FOURTHROOT
U+221D	PROPORTIONAL TO	\propto PROPTO
U+2223	DIVIDES	\mid DIVIDES

U+2224	DOES NOT DIVIDE	\dagger	
U+2225	PARALLEL TO	\parallel	PARALLEL
U+2226	NOT PARALLEL TO	\nparallel	NPARALLEL
U+222B	INTEGRAL	\int	
U+222C	DOUBLE INTEGRAL		
U+222D	TRIPLE INTEGRAL		
U+222E	CONTOUR INTEGRAL	\oint	
U+222F	SURFACE INTEGRAL		
U+2230	VOLUME INTEGRAL		
U+2231	CLOCKWISE INTEGRAL		
U+2232	CLOCKWISE CONTOUR INTEGRAL		
U+2233	ANTICLOCKWISE CONTOUR INTEGRAL		
U+2234	THEREFORE	\therefore	
U+2235	BECAUSE	\because	
U+2236	RATIO		
U+2237	PROPORTION		
U+2239	EXCESS		
U+223A	GEOMETRIC PROPORTION		
U+223B	HOMOTHETIC		
U+223C	TILDE OPERATOR	\sim	
U+223D	REVERSED TILDE	\smile	
U+223E	INVERTED LAZY S		
U+223F	SINE WAVE		
U+2240	WREATH PRODUCT	\wr	WREATH
U+2241	NOT TILDE	\napprox	
U+224B	TRIPLE TILDE		
U+224F	DIFFERENCE BETWEEN	\simeq	BUMPEQ
U+2250	APPROACHES THE LIMIT	\doteq	DOTEQ
U+2258	CORRESPONDS TO		
U+2259	ESTIMATES		
U+225A	EQUIANGULAR TO		
U+225E	MEASURED BY		
U+226C	BETWEEN	\oslash	
U+228C	MULTISET		
U+229A	CIRCLED RING OPERATOR	\odot	CIRCLEDRING
U+229D	CIRCLED DASH	\ominus	
U+22A2	RIGHT TACK	\vdash	VDASH TURNSTILE
U+22A3	LEFT TACK	\dashv	DASHV
U+22A6	ASSERTION	\vDash	
U+22A7	MODELS	\vDash	
U+22A8	TRUE	\vDash	
U+22A9	FORCES	\vDash	
U+22AA	TRIPLE VERTICAL BAR RIGHT TURNSTILE	\vDash	
U+22AB	DOUBLE VERTICAL BAR DOUBLE RIGHT TURNSTILE		
U+22AC	DOES NOT PROVE	\nVdash	
U+22AD	NOT TRUE		
U+22AE	DOES NOT FORCE	\nVdash	
U+22AF	NEGATED DOUBLE VERTICAL BAR DOUBLE RIGHT TURNSTILE	\nVdash	
U+22B6	ORIGINAL OF		
U+22B7	IMAGE OF		
U+22B8	MULTIMAP	\multimap	
U+22B9	HERMITIAN CONJUGATE MATRIX	$\bar{\cdot}$	

U+22BA INTERCALATE
 U+22BE RIGHT ANGLE WITH ARC
 U+22BF RIGHT TRIANGLE
 U+22C0 N-ARY LOGICAL AND
 U+22C1 N-ARY LOGICAL OR
 U+22C2 N-ARY INTERSECTION
 U+22C3 N-ARY UNION
 U+22C4 DIAMOND OPERATOR
 U+22C6 STAR OPERATOR
 U+22C7 DIVISION TIMES
 U+22C8 BOWTIE
 U+22C9 LEFT NORMAL FACTOR SEMIDIRECT PRODUCT
 U+22CA RIGHT NORMAL FACTOR SEMIDIRECT PRODUCT
 U+22CB LEFT SEMIDIRECT PRODUCT
 U+22CC RIGHT SEMIDIRECT PRODUCT
 U+22D4 PITCHFORK
 U+22EE VERTICAL ELLIPSIS
 U+22EF MIDLINE HORIZONTAL ELLIPSIS
 U+22F0 UP RIGHT DIAGONAL ELLIPSIS
 U+22F1 DOWN RIGHT DIAGONAL ELLIPSIS
 U+27C0 THREE DIMENSIONAL ANGLE
 U+27C1 WHITE TRIANGLE CONTAINING SMALL WHITE TRIANGLE
 U+27C2 PERPENDICULAR
 U+27D0 WHITE DIAMOND WITH CENTRED DOT
 U+27D2 ELEMENT OF OPENING UPWARDS
 U+27D3 LOWER RIGHT CORNER WITH DOT
 U+27D4 UPPER LEFT CORNER WITH DOT
 U+27D5 LEFT OUTER JOIN
 U+27D6 RIGHT OUTER JOIN
 U+27D7 FULL OUTER JOIN
 U+27D8 LARGE UP TACK
 U+27D9 LARGE DOWN TACK
 U+27DA LEFT AND RIGHT DOUBLE TURNSTILE
 U+27DB LEFT AND RIGHT TACK
 U+27DC LEFT MULTIMAP
 U+27DD LONG RIGHT TACK
 U+27DE LONG LEFT TACK
 U+27DF UP TACK WITH CIRCLE ABOVE
 U+27E0 LOZENGE DIVIDED BY HORIZONTAL RULE
 U+27E1 WHITE CONCAVE-SIDED DIAMOND
 U+27E2 WHITE CONCAVE-SIDED DIAMOND WITH LEFTWARDS TICK
 U+27E3 WHITE CONCAVE-SIDED DIAMOND WITH RIGHTWARDS TICK
 U+27E4 WHITE SQUARE WITH LEFTWARDS TICK
 U+27E5 WHITE SQUARE WITH RIGHTWARDS TICK
 U+27F0 UPWARDS QUADRUPLE ARROW
 U+27F1 DOWNWARDS QUADRUPLE ARROW
 U+27F2 ANTICLOCKWISE GAPPED CIRCLE ARROW
 U+27F3 CLOCKWISE GAPPED CIRCLE ARROW
 U+27F4 RIGHT ARROW WITH CIRCLED PLUS
 U+27F5 LONG LEFTWARDS ARROW
 U+27F6 LONG RIGHTWARDS ARROW
 U+27F7 LONG LEFT RIGHT ARROW

T

\wedge BIGAND ALL
 \vee BIGOR ANY
 \cap BIGCAP BIGINTERSECT
 \cup BIGCUP BIGUNION
 \diamond DIAMOND
 \star STAR
 $*$
 \boxtimes
 \times
 \times
 \searrow
 \swarrow
 \updownarrow

PERP

U+27F8 LONG LEFTWARDS DOUBLE ARROW
 U+27F9 LONG RIGHTWARDS DOUBLE ARROW
 U+27FA LONG LEFT RIGHT DOUBLE ARROW
 U+27FB LONG LEFTWARDS ARROW FROM BAR
 U+27FC LONG RIGHTWARDS ARROW FROM BAR
 U+27FD LONG LEFTWARDS DOUBLE ARROW FROM BAR
 U+27FE LONG RIGHTWARDS DOUBLE ARROW FROM BAR
 U+27FF LONG RIGHTWARDS SQUIGGLE ARROW
 U+2900 RIGHTWARDS TWO-HEADED ARROW WITH VERTICAL STROKE
 U+2901 RIGHTWARDS TWO-HEADED ARROW WITH DOUBLE VERTICAL STROKE
 U+2902 LEFTWARDS DOUBLE ARROW WITH VERTICAL STROKE
 U+2903 RIGHTWARDS DOUBLE ARROW WITH VERTICAL STROKE
 U+2904 LEFT RIGHT DOUBLE ARROW WITH VERTICAL STROKE
 U+2905 RIGHTWARDS TWO-HEADED ARROW FROM BAR
 U+2906 LEFTWARDS DOUBLE ARROW FROM BAR
 U+2907 RIGHTWARDS DOUBLE ARROW FROM BAR
 U+2908 DOWNWARDS ARROW WITH HORIZONTAL STROKE
 U+2909 UPWARDS ARROW WITH HORIZONTAL STROKE
 U+290A UPWARDS TRIPLE ARROW
 U+290B DOWNWARDS TRIPLE ARROW
 U+290C LEFTWARDS DOUBLE DASH ARROW
 U+290D RIGHTWARDS DOUBLE DASH ARROW
 U+290E LEFTWARDS TRIPLE DASH ARROW
 U+290F RIGHTWARDS TRIPLE DASH ARROW
 U+2910 RIGHTWARDS TWO-HEADED TRIPLE DASH ARROW
 U+2911 RIGHTWARDS ARROW WITH DOTTED STEM
 U+2912 UPWARDS ARROW TO BAR
 U+2913 DOWNWARDS ARROW TO BAR
 U+2914 RIGHTWARDS ARROW WITH TAIL WITH VERTICAL STROKE
 U+2915 RIGHTWARDS ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE
 U+2916 RIGHTWARDS TWO-HEADED ARROW WITH TAIL
 U+2917 RIGHTWARDS TWO-HEADED ARROW WITH TAIL WITH VERTICAL STROKE
 U+2918 RIGHTWARDS TWO-HEADED ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE
 U+2919 LEFTWARDS ARROW-TAIL
 U+291A RIGHTWARDS ARROW-TAIL
 U+291B LEFTWARDS DOUBLE ARROW-TAIL
 U+291C RIGHTWARDS DOUBLE ARROW-TAIL
 U+291D LEFTWARDS ARROW TO BLACK DIAMOND
 U+291E RIGHTWARDS ARROW TO BLACK DIAMOND
 U+291F LEFTWARDS ARROW FROM BAR TO BLACK DIAMOND
 U+2920 RIGHTWARDS ARROW FROM BAR TO BLACK DIAMOND
 U+2921 NORTH WEST AND SOUTH EAST ARROW
 U+2922 NORTH EAST AND SOUTH WEST ARROW
 U+2923 NORTH WEST ARROW WITH HOOK
 U+2924 NORTH EAST ARROW WITH HOOK
 U+2925 SOUTH EAST ARROW WITH HOOK
 U+2926 SOUTH WEST ARROW WITH HOOK
 U+2927 NORTH WEST ARROW AND NORTH EAST ARROW
 U+2928 NORTH EAST ARROW AND SOUTH EAST ARROW
 U+2929 SOUTH EAST ARROW AND SOUTH WEST ARROW
 U+292A SOUTH WEST ARROW AND NORTH WEST ARROW
 U+292B RISING DIAGONAL CROSSING FALLING DIAGONAL

U+292C FALLING DIAGONAL CROSSING RISING DIAGONAL
 U+292D SOUTH EAST ARROW CROSSING NORTH EAST ARROW
 U+292E NORTH EAST ARROW CROSSING SOUTH EAST ARROW
 U+292F FALLING DIAGONAL CROSSING NORTH EAST ARROW
 U+2930 RISING DIAGONAL CROSSING SOUTH EAST ARROW
 U+2931 NORTH EAST ARROW CROSSING NORTH WEST ARROW
 U+2932 NORTH WEST ARROW CROSSING NORTH EAST ARROW
 U+2933 WAVE ARROW POINTING DIRECTLY RIGHT
 U+2934 ARROW POINTING RIGHTWARDS THEN CURVING UPWARDS
 U+2935 ARROW POINTING RIGHTWARDS THEN CURVING DOWNWARDS
 U+2936 ARROW POINTING DOWNWARDS THEN CURVING LEFTWARDS
 U+2937 ARROW POINTING DOWNWARDS THEN CURVING RIGHTWARDS
 U+2938 RIGHT-SIDE ARC CLOCKWISE ARROW
 U+2939 LEFT-SIDE ARC ANTICLOCKWISE ARROW
 U+293A TOP ARC ANTICLOCKWISE ARROW
 U+293B BOTTOM ARC ANTICLOCKWISE ARROW
 U+293C TOP ARC CLOCKWISE ARROW WITH MINUS
 U+293D TOP ARC ANTICLOCKWISE ARROW WITH PLUS
 U+293E LOWER RIGHT SEMICIRCULAR CLOCKWISE ARROW
 U+293F LOWER LEFT SEMICIRCULAR ANTICLOCKWISE ARROW
 U+2940 ANTICLOCKWISE CLOSED CIRCLE ARROW
 U+2941 CLOCKWISE CLOSED CIRCLE ARROW
 U+2942 RIGHTWARDS ARROW ABOVE SHORT LEFTWARDS ARROW
 U+2943 LEFTWARDS ARROW ABOVE SHORT RIGHTWARDS ARROW
 U+2944 SHORT RIGHTWARDS ARROW ABOVE LEFTWARDS ARROW
 U+2945 RIGHTWARDS ARROW WITH PLUS BELOW
 U+2946 LEFTWARDS ARROW WITH PLUS BELOW
 U+2947 RIGHTWARDS ARROW THROUGH X
 U+2948 LEFT RIGHT ARROW THROUGH SMALL CIRCLE
 U+2949 UPWARDS TWO-HEADED ARROW FROM SMALL CIRCLE
 U+294A LEFT BARB UP RIGHT BARB DOWN HARPOON
 U+294B LEFT BARB DOWN RIGHT BARB UP HARPOON
 U+294C UP BARB RIGHT DOWN BARB LEFT HARPOON
 U+294D UP BARB LEFT DOWN BARB RIGHT HARPOON
 U+294E LEFT BARB UP RIGHT BARB UP HARPOON
 U+294F UP BARB RIGHT DOWN BARB RIGHT HARPOON
 U+2950 LEFT BARB DOWN RIGHT BARB DOWN HARPOON
 U+2951 UP BARB LEFT DOWN BARB LEFT HARPOON
 U+2952 LEFTWARDS HARPOON WITH BARB UP TO BAR
 U+2953 RIGHTWARDS HARPOON WITH BARB UP TO BAR
 U+2954 UPWARDS HARPOON WITH BARB RIGHT TO BAR
 U+2955 DOWNWARDS HARPOON WITH BARB RIGHT TO BAR
 U+2956 LEFTWARDS HARPOON WITH BARB DOWN TO BAR
 U+2957 RIGHTWARDS HARPOON WITH BARB DOWN TO BAR
 U+2958 UPWARDS HARPOON WITH BARB LEFT TO BAR
 U+2959 DOWNWARDS HARPOON WITH BARB LEFT TO BAR
 U+295A LEFTWARDS HARPOON WITH BARB UP FROM BAR
 U+295B RIGHTWARDS HARPOON WITH BARB UP FROM BAR
 U+295C UPWARDS HARPOON WITH BARB RIGHT FROM BAR
 U+295D DOWNWARDS HARPOON WITH BARB RIGHT FROM BAR
 U+295E LEFTWARDS HARPOON WITH BARB DOWN FROM BAR
 U+295F RIGHTWARDS HARPOON WITH BARB DOWN FROM BAR

U+2960 UPWARDS HARPOON WITH BARB LEFT FROM BAR
 U+2961 DOWNWARDS HARPOON WITH BARB LEFT FROM BAR
 U+2962 LEFTWARDS HARPOON WITH BARB UP ABOVE LEFTWARDS HARPOON WITH BARB DOWN
 U+2963 UPWARDS HARPOON WITH BARB LEFT BESIDE UPWARDS HARPOON WITH BARB RIGHT
 U+2964 RIGHTWARDS HARPOON WITH BARB UP ABOVE RIGHTWARDS HARPOON WITH BARB DOWN
 U+2965 DOWNWARDS HARPOON WITH BARB LEFT BESIDE DOWNWARDS HARPOON WITH BARB RIGHT
 U+2966 LEFTWARDS HARPOON WITH BARB UP ABOVE RIGHTWARDS HARPOON WITH BARB UP
 U+2967 LEFTWARDS HARPOON WITH BARB DOWN ABOVE RIGHTWARDS HARPOON WITH BARB DOWN
 U+2968 RIGHTWARDS HARPOON WITH BARB UP ABOVE LEFTWARDS HARPOON WITH BARB UP
 U+2969 RIGHTWARDS HARPOON WITH BARB DOWN ABOVE LEFTWARDS HARPOON WITH BARB DOWN
 U+296A LEFTWARDS HARPOON WITH BARB UP ABOVE LONG DASH
 U+296B LEFTWARDS HARPOON WITH BARB DOWN BELOW LONG DASH
 U+296C RIGHTWARDS HARPOON WITH BARB UP ABOVE LONG DASH
 U+296D RIGHTWARDS HARPOON WITH BARB DOWN BELOW LONG DASH
 U+296E UPWARDS HARPOON WITH BARB LEFT BESIDE DOWNWARDS HARPOON WITH BARB RIGHT
 U+296F DOWNWARDS HARPOON WITH BARB LEFT BESIDE UPWARDS HARPOON WITH BARB RIGHT
 U+2970 RIGHT DOUBLE ARROW WITH ROUNDED HEAD
 U+2971 EQUALS SIGN ABOVE RIGHTWARDS ARROW
 U+2972 TILDE OPERATOR ABOVE RIGHTWARDS ARROW
 U+2973 LEFTWARDS ARROW ABOVE TILDE OPERATOR
 U+2974 RIGHTWARDS ARROW ABOVE TILDE OPERATOR
 U+2975 RIGHTWARDS ARROW ABOVE ALMOST EQUAL TO
 U+2976 LESS-THAN ABOVE LEFTWARDS ARROW
 U+2977 LEFTWARDS ARROW THROUGH LESS-THAN
 U+2978 GREATER-THAN ABOVE RIGHTWARDS ARROW
 U+2979 SUBSET ABOVE RIGHTWARDS ARROW
 U+297A LEFTWARDS ARROW THROUGH SUBSET
 U+297B SUPerset ABOVE LEFTWARDS ARROW
 U+297C LEFT FISH TAIL
 U+297D RIGHT FISH TAIL
 U+297E UP FISH TAIL
 U+297F DOWN FISH TAIL
 U+2980 TRIPLE VERTICAL BAR DELIMITER
 U+2981 Z NOTATION SPOT
 U+2982 Z NOTATION TYPE COLON
 U+2999 DOTTED FENCE
 U+299A VERTICAL ZIGZAG LINE
 U+299B MEASURED ANGLE OPENING LEFT
 U+299C RIGHT ANGLE VARIANT WITH SQUARE
 U+299D MEASURED RIGHT ANGLE WITH DOT
 U+299E ANGLE WITH S INSIDE
 U+299F ACUTE ANGLE
 U+29A0 SPHERICAL ANGLE OPENING LEFT
 U+29A1 SPHERICAL ANGLE OPENING UP
 U+29A2 TURNED ANGLE
 U+29A3 REVERSED ANGLE
 U+29A4 ANGLE WITH UNDERBAR
 U+29A5 REVERSED ANGLE WITH UNDERBAR
 U+29A6 OBLIQUE ANGLE OPENING UP
 U+29A7 OBLIQUE ANGLE OPENING DOWN
 U+29A8 MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING UP AND RIGHT
 U+29A9 MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING UP AND LEFT

U+29AA MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING DOWN AND RIGHT
 U+29AB MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING DOWN AND LEFT
 U+29AC MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING RIGHT AND UP
 U+29AD MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING LEFT AND UP
 U+29AE MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING RIGHT AND DOWN
 U+29AF MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING LEFT AND DOWN
 U+29B0 REVERSED EMPTY SET
 U+29B1 EMPTY SET WITH OVERBAR
 U+29B2 EMPTY SET WITH SMALL CIRCLE ABOVE
 U+29B3 EMPTY SET WITH RIGHT ARROW ABOVE
 U+29B4 EMPTY SET WITH LEFT ARROW ABOVE
 U+29B5 CIRCLE WITH HORIZONTAL BAR
 U+29B6 CIRCLED VERTICAL BAR
 U+29B7 CIRCLED PARALLEL
 U+29B9 CIRCLED PERPENDICULAR
 U+29BA CIRCLE DIVIDED BY HORIZONTAL BAR AND TOP HALF DIVIDED BY VERTICAL BAR
 U+29BB CIRCLE WITH SUPERIMPOSED X
 U+29BD UP ARROW THROUGH CIRCLE
 U+29BE CIRCLED WHITE BULLET
 U+29BF CIRCLED BULLET
 U+29C2 CIRCLE WITH SMALL CIRCLE TO THE RIGHT
 U+29C3 CIRCLE WITH TWO HORIZONTAL STROKES TO THE RIGHT
 U+29C5 SQUARED FALLING DIAGONAL SLASH
 U+29C7 SQUARED SMALL CIRCLE
 U+29C8 SQUARED SQUARE
 U+29C9 TWO JOINED SQUARES
 U+29CA TRIANGLE WITH DOT ABOVE
 U+29CB TRIANGLE WITH UNDERBAR
 U+29CC S IN TRIANGLE
 U+29CD TRIANGLE WITH SERIFS AT BOTTOM
 U+29CE RIGHT TRIANGLE ABOVE LEFT TRIANGLE
 U+29CF LEFT TRIANGLE BESIDE VERTICAL BAR
 U+29D0 VERTICAL BAR BESIDE RIGHT TRIANGLE
 U+29D1 BOWTIE WITH LEFT HALF BLACK
 U+29D2 BOWTIE WITH RIGHT HALF BLACK
 U+29D3 BLACK BOWTIE
 U+29D6 WHITE HOURGLASS
 U+29D7 BLACK HOURGLASS
 U+29DC INCOMPLETE INFINITY
 U+29DD TIE OVER INFINITY
 U+29DE INFINITY NEGATED WITH VERTICAL BAR
 U+29DF DOUBLE-ENDED MULTIMAP
 U+29E0 SQUARE WITH CONTOURED OUTLINE
 U+29E1 INCREASES AS
 U+29E2 SHUFFLE PRODUCT
 U+29E6 GLEICH STARK
 U+29E7 THERMODYNAMIC
 U+29E8 DOWN-POINTING TRIANGLE WITH LEFT HALF BLACK
 U+29E9 DOWN-POINTING TRIANGLE WITH RIGHT HALF BLACK
 U+29EA BLACK DIAMOND WITH DOWN ARROW
 U+29EB BLACK LOZENGE
 U+29EC WHITE CIRCLE WITH DOWN ARROW

U+29ED	BLACK CIRCLE WITH DOWN ARROW	
U+29EE	ERROR-BARRED WHITE SQUARE	
U+29EF	ERROR-BARRED BLACK SQUARE	
U+29F0	ERROR-BARRED WHITE DIAMOND	
U+29F1	ERROR-BARRED BLACK DIAMOND	
U+29F2	ERROR-BARRED WHITE CIRCLE	
U+29F3	ERROR-BARRED BLACK CIRCLE	
U+29F4	RULE-DELAYED	
U+29F6	SOLIDUS WITH OVERBAR	
U+29F7	REVERSE SOLIDUS WITH HORIZONTAL STROKE	
U+29FA	DOUBLE PLUS	
U+29FB	TRIPLE PLUS	
U+29FE	TINY	
U+29FF	MINY	
U+2A00	N-ARY CIRCLED DOT OPERATOR	\odot BIGODOT
U+2A01	N-ARY CIRCLED PLUS OPERATOR	\oplus BIGOPLUS
U+2A02	N-ARY CIRCLED TIMES OPERATOR	\otimes BIGOTIMES
U+2A03	N-ARY UNION OPERATOR WITH DOT	BIGUDOT
U+2A04	N-ARY UNION OPERATOR WITH PLUS	BIGUPLUS
U+2A05	N-ARY SQUARE INTERSECTION OPERATOR	BIGSQCAP
U+2A06	N-ARY SQUARE UNION OPERATOR	BIGSQCUP
U+2A07	TWO LOGICAL AND OPERATOR	
U+2A08	TWO LOGICAL OR OPERATOR	
U+2A09	N-ARY TIMES OPERATOR	BIGTIMES
U+2A0A	MODULO TWO SUM	
U+2A10	CIRCULATION FUNCTION	
U+2A11	ANTICLOCKWISE INTEGRATION	
U+2A12	LINE INTEGRATION WITH RECTANGULAR PATH AROUND POLE	
U+2A13	LINE INTEGRATION WITH SEMICIRCULAR PATH AROUND POLE	
U+2A14	LINE INTEGRATION NOT INCLUDING THE POLE	
U+2A1D	JOIN	\bowtie JOIN
U+2A1E	LARGE LEFT TRIANGLE OPERATOR	
U+2A1F	Z NOTATION SCHEMA COMPOSITION	
U+2A20	Z NOTATION SCHEMA PIPING	
U+2A21	Z NOTATION SCHEMA PROJECTION	
U+2A32	SEMIDIRECT PRODUCT WITH BOTTOM CLOSED	
U+2A33	SMASH PRODUCT	
U+2A3C	INTERIOR PRODUCT	
U+2A3D	RIGHTHAND INTERIOR PRODUCT	
U+2A3E	Z NOTATION RELATIONAL COMPOSITION	
U+2A3F	AMALGAMATION OR COPRODUCT	
U+2A57	SLOPING LARGE OR	
U+2A58	SLOPING LARGE AND	
U+2A61	SMALL VEE WITH UNDERBAR	
U+2A64	Z NOTATION DOMAIN ANTIRESTRICTION	
U+2A65	Z NOTATION RANGE ANTIRESTRICTION	
U+2A68	TRIPLE HORIZONTAL BAR WITH DOUBLE VERTICAL STROKE	
U+2A69	TRIPLE HORIZONTAL BAR WITH TRIPLE VERTICAL STROKE	
U+2A6A	TILDE OPERATOR WITH DOT ABOVE	
U+2A6B	TILDE OPERATOR WITH RISING DOTS	
U+2A6D	CONGRUENT WITH DOT ABOVE	
U+2ACD	SQUARE LEFT OPEN BOX OPERATOR	

U+2ACE SQUARE RIGHT OPEN BOX OPERATOR
U+2AD9 ELEMENT OF OPENING DOWNWARDS
U+2ADA PITCHFORK WITH TEE TOP
U+2ADC FORKING
U+2ADD NONFORKING
U+2ADE SHORT LEFT TACK
U+2ADF SHORT DOWN TACK
U+2AE0 SHORT UP TACK
U+2AE1 PERPENDICULAR WITH S
U+2AE2 VERTICAL BAR TRIPLE RIGHT TURNSTILE
U+2AE3 DOUBLE VERTICAL BAR LEFT TURNSTILE
U+2AE4 VERTICAL BAR DOUBLE LEFT TURNSTILE
U+2AE5 DOUBLE VERTICAL BAR DOUBLE LEFT TURNSTILE
U+2AE6 LONG DASH FROM LEFT MEMBER OF DOUBLE VERTICAL
U+2AE7 SHORT DOWN TACK WITH OVERBAR
U+2AE8 SHORT UP TACK WITH UNDERBAR
U+2AE9 SHORT UP TACK ABOVE SHORT DOWN TACK
U+2AEA DOUBLE DOWN TACK
U+2AEB DOUBLE UP TACK
U+2AEC DOUBLE STROKE NOT SIGN
U+2AED REVERSED DOUBLE STROKE NOT SIGN
U+2AEE DOES NOT DIVIDE WITH REVERSED NEGATION SLASH
U+2AEF VERTICAL LINE WITH CIRCLE ABOVE
U+2AF0 VERTICAL LINE WITH CIRCLE BELOW
U+2AF1 DOWN TACK WITH CIRCLE BELOW
U+2AF2 PARALLEL WITH HORIZONTAL STROKE
U+2AF3 PARALLEL WITH TILDE OPERATOR
U+2AF5 TRIPLE VERTICAL BAR WITH HORIZONTAL STROKE
U+2AF6 TRIPLE COLON OPERATOR
U+2AFB TRIPLE SOLIDUS BINARY RELATION
U+2AFC LARGE TRIPLE VERTICAL BAR OPERATOR
U+2AFE WHITE VERTICAL BAR
U+2AFF N-ARY WHITE VERTICAL BAR

Appendix G

Simplified Grammar for Application Programmers and Library Writers

In this chapter, we provide a simplified grammar of the Fortress concrete syntax as documentation, to enable Fortress programmers to understand it more easily. The full Fortress grammar is described in Appendix H.

The simplified grammar does not describe the details of the uses of operators, whitespaces, and semicolons. Instead, they are described as follows according to three different contexts influencing the whitespace-sensitivity of expressions:

statement Expressions immediately enclosed by a block expression are in a statement-like context. Multiple expressions can appear on a line if they are separated (or terminated) by semicolons. If an expression can legally end at the end of a line, it does; if it cannot, it does not. A prefix or infix operator that lacks its last operand prevents an expression from ending. For example,

```
an = expression+  
      spanning+  
      four+  
      lines  
a = oneLiner  
four(); on(); one(); line();
```

nested An expression or list of expressions immediately enclosed by parentheses or braces is nested. Multiple expressions are separated by commas, and the end of a line does not end an expression. Because of this effect, the meaning of a several lines of code can change if they are wrapped in parentheses. Parentheses can also be used to ensure that a multiline expression is not terminated prematurely without paying special attention to line endings.

```
lhs = rhs  
– aSeparateExpression  
postProfit(revenue  
             – expenses)
```

pasted Fortress has special syntax for matrix pasting. Within square brackets, whitespace-separated expressions are treated (depending on their type) as either matrix elements or submatrices within a row. Because whitespace is the separator, it also ends expressions where possible. In addition, newline-or-semicolon-separated rows are pasted vertically along their columns. Higher-dimensional pasting is expressed with repeated semicolons, but repeated newlines do not have the same effect.

$$\begin{aligned}
id2a &= [1\ 0; 0\ 1] \\
id2b &= [1\ 0; \\
&\quad 0\ 1] \\
id2c &= [1\ 0 \\
&\quad 0\ 1] \\
cube_2 &= [1\ 0; 0\ 1; ; 1\ -1; 1\ 1]
\end{aligned}$$

A restricted form of the pasting syntax can also be used on the left hand side of variable declarations to express both declaration and submatrix decomposition.

$$\begin{aligned}
\begin{bmatrix} top \\ bot \end{bmatrix} &= X \\
\begin{bmatrix} left\ right \end{bmatrix} &= Y \\
Z &= \begin{bmatrix} top \cdot left\ top \cdot right; \\ bot \cdot left\ bot \cdot right \end{bmatrix}
\end{aligned}$$

Section 6.5 describes matrix unpadding in detail and includes more examples.

The simplified grammar is written in the extended BNF form with the following three postfix operators:

- *?*: which means that its argument (the symbol or group of symbols in parentheses to the left of the operator) can appear zero or one time
- ***: which means that its argument can appear zero or more times
- *+*: which means that its argument can appear one or more times

G.1 Components and APIs

```

File           ::= CompilationUnit
                | Imports? Exports Decls?
                | Imports? AbsDecls
                | Imports AbsDecls?
CompilationUnit ::= Component
                | Api
Component      ::= component DottedId Imports? Exports Decls? end
Api            ::= api DottedId Imports? AbsDecls? end
Imports        ::= Import+
Import         ::= import ImportFrom
                | import AliasedDottedIds
ImportFrom     ::= * (except Names)? from DottedId
                | AliasedNames from DottedId
Names          ::= Name
                | { NameList }
NameList       ::= Name( , Name)*
AliasedNames   ::= AliasedName
                | { AliasedNameList }
AliasedName    ::= Id ( as DottedId)?
                | opr Op ( as Op)?
                | opr LeftEncloser RightEncloser ( as LeftEncloser RightEncloser)?
AliasedNameList ::= AliasedName( , AliasedName)*

```

```

AliasedDottedIds ::= AliasedDottedId
                  | { AliasedDottedIdList }
AliasedDottedId ::= DottedId ( as DottedId )?
AliasedDottedIdList ::= AliasedDottedId ( , AliasedDottedId )*
Exports ::= Export+
Export ::= export DottedIds
DottedIds ::= DottedId
            | { DottedIdList }
DottedIdList ::= DottedId ( , DottedId )*

```

G.2 Top-level Declarations

```

Decls ::= Decl+
Decl ::= TraitDecl
        | ObjectDecl
        | VarDecl
        | FnDecl
        | DimUnitDecl
        | TypeAlias
        | TestDecl
        | PropertyDecl
        | ExternalSyntax
AbsDecls ::= AbsDecl+
AbsDecl ::= AbsTraitDecl
            | AbsObjectDecl
            | AbsVarDecl
            | AbsFnDecl
            | DimUnitDecl
            | TypeAlias
            | TestDecl
            | PropertyDecl
            | AbsExternalSyntax

```

G.3 Trait and Object Declarations

A *TraitHeader* may have 0 or 1 of each *TraitClause*.

```

TraitDecl ::= TraitHeader GoInATrait? end
TraitHeader ::= TraitMods? trait Id StaticParams? Extends? TraitClauses?
TraitClauses ::= TraitClause+
TraitClause ::= Excludes
                | Comprises
                | Where
GoInATrait ::= GoFrontInATrait GoBackInATrait?
                | GoBackInATrait

```

```

GoFrontInATrait ::= GoesFrontInATrait+
GoesFrontInATrait ::= AbsFldDecl
| GetterSetterDecl
| PropertyDecl
GoBackInATrait ::= GoesBackInATrait+
GoesBackInATrait ::= MdDecl
| PropertyDecl
ObjectDecl ::= ObjectHeader GoInAnObject? end
ObjectHeader ::= ObjectMods? object Id StaticParams? ObjectValParam? Extends? FnClauses
ObjectValParam ::= ( ObjectParams? )
ObjectParams ::= (ObjectParam ,)* (ObjectVarargs ,)? ObjectKeyword( , ObjectKeyword)*
| (ObjectParam ,)* ObjectVarargs
| ObjectParam ( , ObjectParam)*
ObjectVarargs ::= transient Varargs
ObjectKeyword ::= ObjectParam = Expr
ObjectParam ::= FldMods? Param
| transient Param
GoInAnObject ::= GoFrontInAnObject GoBackInAnObject?
| GoBackInAnObject
GoFrontInAnObject ::= GoesFrontInAnObject+
GoesFrontInAnObject ::= FldDecl
| GetterSetterDef
| PropertyDecl
GoBackInAnObject ::= GoesBackInAnObject+
GoesBackInAnObject ::= MdDef
| PropertyDecl
AbsTraitDecl ::= TraitHeader AbsGoInATrait? end
AbsGoInATrait ::= AbsGoFrontInATrait AbsGoBackInATrait?
| AbsGoBackInATrait
AbsGoFrontInATrait ::= AbsGoesFrontInATrait+
AbsGoesFrontInATrait ::= ApiFldDecl
| AbsGetterSetterDecl
| PropertyDecl
AbsGoBackInATrait ::= AbsGoesBackInATrait+
AbsGoesBackInATrait ::= AbsMdDecl
| AbsCoercion
| PropertyDecl
AbsObjectDecl ::= ObjectHeader AbsGoInAnObject? end
AbsGoInAnObject ::= AbsGoFrontInAnObject AbsGoBackInAnObject?
| AbsGoBackInAnObject
AbsGoFrontInAnObject ::= AbsGoesFrontInAnObject+
AbsGoesFrontInAnObject ::= ApiFldDecl
| AbsGetterSetterDecl
| PropertyDecl
AbsGoBackInAnObject ::= AbsGoesBackInAnObject+
AbsGoesBackInAnObject ::= AbsMdDecl
| AbsCoercion
| PropertyDecl

```

G.4 Variable Declarations

```
VarDecl ::= VarWTypes InitVal
          | VarWoTypes = Expr
          | VarWoTypes : TypeRef ... InitVal
          | VarWoTypes : SimpleTupleType InitVal
VarWTypes ::= VarWType
             | ( VarWType(, VarWType)+ )
VarWType ::= VarMods? Id IsType
VarWoTypes ::= VarWoType
              | ( VarWoType(, VarWoType)+ )
VarWoType ::= VarMods? Id
InitVal ::= (= | :=) Expr
AbsVarDecl ::= VarWTypes
               | VarWoTypes : TypeRef ...
               | VarWoTypes : SimpleTupleType
```

G.5 Function Declarations

```
FnDecl ::= FnDef
          | AbsFnDecl
FnDef ::= FnMods? FnHeaderFront FnHeaderClause = Expr
AbsFnDecl ::= FnMods? FnHeaderFront FnHeaderClause
              | Name : ArrowType
FnHeaderFront ::= Id StaticParams? ValParam
                  | OpHeaderFront
```

G.6 Dimension, Unit, Type Alias, Test, Property, and External Syntax Declarations

```
DimUnitDecl ::= dim Id (= DimRef)? (default Id)?
              | (unit | SI_unit) Id+ (: DimRef)? (= Expr)?
              | dim Id (= DimRef)? (unit | SI_unit) Id+ (= Expr)?
TypeAlias ::= type Id StaticParams? = TypeRef
TestDecl ::= test Id [GeneratorList] = Expr
PropertyDecl ::= property (Id =)? (∀ ValParam)? Expr
ExternalSyntax ::= syntax OpenExpander Id CloseExpander = Expr
OpenExpander ::= Id
                 | LeftEncloser
CloseExpander ::= Id
                  | RightEncloser
                  | end
AbsExternalSyntax ::= syntax OpenExpander Id CloseExpander
```

G.7 Headers

Each modifier should not appear multiple times in a declaration.

```

Extends ::= extends TraitTypes
Excludes ::= excludes TraitTypes
Comprises ::= comprises ComprisingTypes
TraitTypes ::= TraitType
| { TraitTypeList }
TraitTypeList ::= TraitType(, TraitType)*
ComprisingTypes ::= TraitType
| { ComprisingTypeList }
ComprisingTypeList ::= ...
| TraitType(, TraitType)* (, ...)?
Where ::= where { WhereClauseList }
WhereClauseList ::= WhereClause(, WhereClause)*
WhereClause ::= Id Extends
| TypeAlias
| NatConstraint
| IntConstraint
| BoolConstraint
| UnitConstraint
| TypeRef coerces TypeRef
| TypeRef widens TypeRef
NatConstraint ::= nat Id
| NatRef < NatRef
| NatRef ≤ NatRef
| NatRef > NatRef
| NatRef ≥ NatRef
| NatRef = NatRef
| (NatConstraint)
NatRef ::= StaticArg
IntConstraint ::= int Id
| NatRef < NatRef
| NatRef ≤ NatRef
| NatRef > NatRef
| NatRef ≥ NatRef
| NatRef = NatRef
| (IntConstraint)
BoolConstraint ::= bool Id
| NOT BoolRef
| BoolRef OR BoolRef
| BoolRef AND BoolRef
| BoolRef IMPLIES BoolRef
| BoolRef = BoolRef
| (BoolConstraint)
BoolRef ::= StaticArg
UnitConstraint ::= unit Id
| UnitRef = UnitRef
FnHeaderClause ::= IsType? FnClauses
FnClauses ::= Throws? Where? Contract
Throws ::= throws MayTraitTypes
MayTraitTypes ::= {}
| TraitTypes

```


<i>Contract</i>	::=	<i>Requires? Ensures? Invariant?</i>
<i>Requires</i>	::=	<i>requires</i> { <i>ExprList?</i> }
<i>Ensures</i>	::=	<i>ensures</i> { <i>ProvidedList?</i> }
<i>ProvidedList</i>	::=	<i>Provided</i> (, <i>Provided</i>)*
<i>Provided</i>	::=	<i>Expr</i> (<i>provided Expr</i>)?
<i>Invariant</i>	::=	<i>invariant</i> { <i>ExprList?</i> }
<i>CoercionClauses</i>	::=	<i>Throws? CoercionWhere? Contract</i>
<i>CoercionWhere</i>	::=	<i>where</i> { <i>CoercionWhereClauseList</i> }
<i>CoercionWhereClauseList</i>	::=	<i>CoercionWhereClause</i> (, <i>CoercionWhereClause</i>)*
<i>CoercionWhereClause</i>	::=	<i>WhereClause</i> <i>TypeRef</i> widens or coerces <i>TypeRef</i>
<i>UniversalMod</i>	::=	<i>private</i> <i>test</i>
<i>TraitMod</i>	::=	<i>value</i> <i>UniversalMod</i>
<i>TraitMods</i>	::=	<i>TraitMod</i> ⁺
<i>ObjectMods</i>	::=	<i>TraitMods</i>
<i>FnMod</i>	::=	<i>LocalFnMod</i> <i>UniversalMod</i>
<i>FnMods</i>	::=	<i>FnMod</i> ⁺
<i>VarMod</i>	::=	<i>var</i> <i>UniversalMod</i>
<i>VarMods</i>	::=	<i>VarMod</i> ⁺
<i>AbsFldMod</i>	::=	<i>hidden</i> <i>settable</i> <i>wrapped</i> <i>UniversalMod</i>
<i>AbsFldMods</i>	::=	<i>AbsFldMod</i> ⁺
<i>FldMod</i>	::=	<i>var</i> <i>AbsFldMod</i>
<i>FldMods</i>	::=	<i>FldMod</i> ⁺
<i>ApiFldMod</i>	::=	<i>hidden</i> <i>settable</i> <i>UniversalMod</i>
<i>ApiFldMods</i>	::=	<i>ApiFldMod</i> ⁺
<i>LocalFnMod</i>	::=	<i>atomic</i> <i>io</i>
<i>LocalFnMods</i>	::=	<i>LocalFnMod</i> ⁺
<i>StaticParams</i>	::=	[[<i>StaticParamList</i>]]
<i>StaticParamList</i>	::=	<i>StaticParam</i> (, <i>StaticParam</i>)*
<i>StaticParam</i>	::=	<i>Id Extends?</i> (<i>absorbs unit</i>)? <i>nat Id</i> <i>int Id</i> <i>bool Id</i> <i>dim Id</i> <i>unit Id</i> (: <i>DimRef</i>)? (<i>absorbs unit</i>)? <i>opr Op</i> <i>ident Id</i>

G.8 Parameters

<i>ValParam</i>	::=	<i>BindId</i> (<i>Params?</i>)
<i>Params</i>	::=	(<i>Param</i> ,)* (<i>Varargs</i> ,)? <i>Keyword</i> (, <i>Keyword</i>)* (<i>Param</i> ,)* <i>Varargs</i> <i>Param</i> (, <i>Param</i>)*
<i>VarargsParam</i>	::=	<i>Id</i> : <i>TypeRef</i> ...
<i>Varargs</i>	::=	<i>VarargsParam</i>
<i>Keyword</i>	::=	<i>Param</i> = <i>Expr</i>
<i>PlainParam</i>	::=	<i>BindId IsType?</i> <i>TypeRef</i>
<i>Param</i>	::=	<i>PlainParam</i>

```

OpHeaderFront ::= opr StaticParams? (LeftEncloser | Encloser) Params (RightEncloser | Encloser)
                ( := ( SubscriptAssignParam ) )?
                | opr StaticParams? ValParam Op
                | opr (Op | Encloser) StaticParams? ValParam
SubscriptAssignParam ::= Varargs
                | Param

```

G.9 Method Declarations

```

MdDecl ::= MdDef
          | AbsMdDecl
MdDef ::= FnMods? MdHeaderFront FnHeaderClause = Expr
          | Coercion
AbsMdDecl ::= abstract ? FnMods? MdHeaderFront FnHeaderClause
MdHeaderFront ::= (Receiver .)? Id StaticParams? MdValParam
                | OpHeaderFront
Receiver ::= Id
            | self
GetterSetterDecl ::= GetterSetterDef
                    | AbsGetterSetterDecl
GetterSetterDef ::= FnMods? GetterSetterMod MdHeaderFront FnHeaderClause = Expr
GetterSetterMod ::= getter | setter
AbsGetterSetterDecl ::= abstract ? FnMods? GetterSetterMod MdHeaderFront FnHeaderClause
Coercion ::= widening ? coercion StaticParams? (Id IsType) CoercionClauses = Expr
AbsCoercion ::= widening ? coercion StaticParams? (Id IsType) CoercionClauses

```

G.10 Method Parameters

```

MdValParam ::= ( MdParams? )
MdParams ::= (MdParam , ) * (Varargs , ) ? MdKeyword ( , MdKeyword ) *
            | (MdParam , ) * Varargs
            | MdParam ( , MdParam ) *
MdKeyword ::= MdParam = Expr
MdParam ::= Param
            | self

```

G.11 Field Declarations

```

FldDecl ::= FldWTypes InitVal
            | FldWoTypes = Expr
            | FldWoTypes : TypeRef ... InitVal
            | FldWoTypes : SimpleTupleType InitVal
FldWTypes ::= FldWType
            | ( FldWType ( , FldWType ) + )
FldWType ::= FldMods? Id IsType
FldWoTypes ::= FldWoType
            | ( FldWoType ( , FldWoType ) + )
FldWoType ::= FldMods? Id

```

G.12 Abstract Field Declarations

```

AbsFldDecl ::= AbsFldWTypes
              | AbsFldWoTypes : TypeRef ...
              | AbsFldWoTypes : SimpleTupleType
AbsFldWTypes ::= AbsFldWType
                  | ( AbsFldWType( , AbsFldWType)+ )
AbsFldWType ::= AbsFldMods? Id IsType
AbsFldWoTypes ::= AbsFldWoType
                   | ( AbsFldWoType( , AbsFldWoType)+ )
AbsFldWoType ::= AbsFldMods? Id
ApiFldDecl ::= ApiFldMods? Id IsType

```

G.13 Expressions

```

Expr ::= AssignLefts AssignOp Expr
          | OpExpr
          | DelimitedExpr
          | FlowExpr
          | fn ValParam IsType? Throws? ⇒ Expr
          | Expr as TypeRef
          | Expr asif TypeRef
          | UnitExpr
UnitExpr ::= UnitRef
              | Expr UnitRef
              | Expr · UnitRef
              | Expr / UnitRef
              | Expr per UnitRef
              | Expr in UnitRef
UnitRef ::= StaticArg
AssignLefts ::= ( AssignLeft( , AssignLeft)* )
                | AssignLeft
AssignLeft ::= SubscriptExpr
                | FieldSelection
                | BindId
SubscriptExpr ::= Primary [ExprList?]
FieldSelection ::= Primary . Id
OpExpr ::= EncloserOp OpExpr? EncloserOp?
            | OpExpr EncloserOp OpExpr?
            | Primary
EncloserOp ::= Encloser
                | Op

```

```

Primary ::= Comprehension
| Id[StaticArgList ]
| BaseExpr
| LeftEncloser ExprList? RightEncloser
| Primary [ExprList?]
| Primary . Id ([StaticArgList ])? TupleExpr
| Primary . Id ([StaticArgList ])? ()
| Primary . Id
| Primary ^ BaseExpr
| Primary ExponentOp
| Primary TupleExpr
| Primary ()
| Primary Primary
| TraitType . coercion ([StaticArgList ])? (Expr)

FlowExpr ::= exit Id? (with Expr)?
| Accumulator ([ GeneratorList ])? Expr
| atomic AtomicBack
| tryatomic AtomicBack
| spawn Expr
| throw Expr

AtomicBack ::= AssignLefts AssignOp Expr
| OpExpr
| DelimitedExpr

GeneratorList ::= Generator( , Generator)*
Generator ::= Id ← Expr
| ( Id , IdList ) ← Expr
| Expr

```

G.14 Expressions Enclosed by Keywords

```

DelimitedExpr ::= TupleExpr
| object Extends? GoInAnObject end
| Do
| label Id BlockElems end Id
| while Expr Do
| for GeneratorList DoFront end
| if Expr then BlockElems Elifs? Else? end
| ( if Expr then BlockElems Elifs? Else end?)
| case Expr Op? of CaseClauses CaseElse? end
| case (largest | smallest) Op? of CaseClauses end
| typecase TypecaseBindings of TypecaseClauses CaseElse? end
| try BlockElems Catch? (forbid TraitTypes)? (finally BlockElems)? end

Do ::= (DoFront also)* DoFront end
DoFront ::= (at Expr)? atomic? do BlockElems?
TupleExpr ::= ( (Expr ,)* (Expr ... ,)? Binding ( , Binding)* )
| NoKeyTuple
NoKeyTuple ::= ( (Expr ,)* Expr ... )
| ( (Expr ,)* Expr )

```

<i>Elifs</i>	::=	<i>Elif</i> ⁺
<i>Elif</i>	::=	<code>elif Expr then BlockElems</code>
<i>Else</i>	::=	<code>else BlockElems</code>
<i>CaseClauses</i>	::=	<i>CaseClause</i> ⁺
<i>CaseClause</i>	::=	<code>Expr ⇒ BlockElems</code>
<i>CaseElse</i>	::=	<code>else ⇒ BlockElems</code>
<i>TypecaseBindings</i>	::=	(<i>BindingList</i>) <i>Binding</i> <i>Id</i>
<i>BindingList</i>	::=	<i>Binding</i> (, <i>Binding</i>) [*]
<i>Binding</i>	::=	<code>BindId = Expr</code>
<i>TypecaseClauses</i>	::=	<i>TypecaseClause</i> ⁺
<i>TypecaseClause</i>	::=	<i>TypecaseTypeRefs</i> ⇒ <i>BlockElems</i>
<i>TypecaseTypeRefs</i>	::=	(<i>TypeRefList</i>) <i>TypeRef</i>
<i>Catch</i>	::=	<code>catch Id CatchClauses</code>
<i>CatchClauses</i>	::=	<i>CatchClause</i> ⁺
<i>CatchClause</i>	::=	<i>TraitType</i> ⇒ <i>BlockElems</i>
<i>Comprehension</i>	::=	{ <i>Expr</i> <i>GeneratorList</i> } { <i>Entry</i> <i>GeneratorList</i> } ⟨ <i>Expr</i> <i>GeneratorList</i> ⟩ [<i>ArrayComprehensionClause</i> ⁺]
<i>Entry</i>	::=	<i>Expr</i> ↦ <i>Expr</i>
<i>ArrayComprehensionLeft</i>	::=	<i>IdOrInt</i> ↦ <i>Expr</i> (<i>IdOrInt</i> , <i>IdOrIntList</i>) ↦ <i>Expr</i>
<i>ArrayComprehensionClause</i>	::=	<i>ArrayComprehensionLeft</i> <i>GeneratorList</i>
<i>IdOrInt</i>	::=	<i>Id</i> <i>IntLiteral</i>
<i>IdOrIntList</i>	::=	<i>IdOrInt</i> (, <i>IdOrInt</i>) [*]
<i>BaseExpr</i>	::=	<i>NoKeyTuple</i> <i>Literal</i> <i>Id</i> <code>self</code>
<i>ExprList</i>	::=	<i>Expr</i> (, <i>Expr</i>) [*]

G.15 Local Declarations

<i>BlockElems</i>	::=	<i>BlockElem</i> ⁺
<i>BlockElem</i>	::=	<i>LocalVarFnDecl</i> <i>Expr</i> (, <i>GeneratorList</i>)?
<i>LocalVarFnDecl</i>	::=	<i>LocalFnDecl</i> ⁺ <i>LocalVarDecl</i>
<i>LocalFnDecl</i>	::=	<i>LocalFnMods</i> ? <i>FnHeaderFront</i> <i>FnHeaderClause</i> = <i>Expr</i>
<i>LocalVarDecl</i>	::=	<i>LocalVarWTypes</i> <i>InitVal</i> <i>LocalVarWTypes</i> <i>LocalVarWoTypes</i> = <i>Expr</i> <i>LocalVarWoTypes</i> : <i>TypeRef</i> ... <i>InitVal</i> ? <i>LocalVarWoTypes</i> : <i>SimpleTupleType</i> <i>InitVal</i> ?
<i>LocalVarWTypes</i>	::=	<i>LocalVarWType</i> (<i>LocalVarWType</i> (, <i>LocalVarWType</i>) ⁺)
<i>LocalVarWType</i>	::=	<code>var ? Id IsType</code>

LocalVarWoTypes ::= *LocalVarWoType*
| (*LocalVarWoType*(, *LocalVarWoType*)⁺)
LocalVarWoType ::= *var* ? *Id*
| *Unpasting*
Unpasting ::= [*UnpastingElems*]
UnpastingElems ::= *UnpastingElem* *RectSeparator* *UnpastingElems*
| *UnpastingElem*
UnpastingElem ::= *Id* ([*UnpastingDim*])?
| *Unpasting*
UnpastingDim ::= *ExtentRange* (× *ExtentRange*)⁺

G.16 Literals

Literal ::= ()
| *NumericLiteral*
| *CharLiteral*
| *StringLiteral*
| { *Entry* (, *Entry*)^{*} }
| [*RectElements*]
RectElements ::= *Expr* *MultiDimCons*^{*}
MultiDimCons ::= *RectSeparator* *Expr*
RectSeparator ::= ;+
| *Whitespace*

G.17 Types

IsType ::= : *TypeRef*
TypeRef : ::= *TraitType*
| *ArrowType*
| *TupleType*
| (*TypeRef*?)
| *DimType*
TraitType ::= *DottedId* ([*StaticArgList*])?
| { *TypeRef* ↦ *TypeRef* }
| ⟨ *TypeRef* ⟩
| *TypeRef* [*ArraySize*?]
| *TypeRef* ^ *StaticArg*
| *TypeRef* ^ (*ExtentRange* (× *ExtentRange*)^{*})
ArrowType ::= *TypeRef* → *TypeRef* *Throws*?
DimType ::= *DimRef*
| *TypeRef* *DimRef*
| *TypeRef* · *DimRef*
| *TypeRef* / *DimRef*
| *TypeRef* per *DimRef*
| *TypeRef* *UnitRef*
| *TypeRef* · *UnitRef*
| *TypeRef* / *UnitRef*
| *TypeRef* per *UnitRef*
| *TypeRef* in *DimRef*
DimRef ::= *StaticArg*

TupleType ::= ((*TypeRef* ,)* (*TypeRef* ... ,)? *KeywordType*(, *KeywordType*)*)
| ((*TypeRef* ,)* *TypeRef* ...)
| *SimpleTupleType*
KeywordType ::= *Id* = *TypeRef*
SimpleTupleType ::= (*TypeRef* , *TypeRefList*)
TypeRefList ::= *TypeRef*(, *TypeRef*)*
StaticArgList ::= *StaticArg*(, *StaticArg*)*
StaticArg ::= *Number*
| *Op*
| *true*
| *false*
| *Unity*
| *dimensionless*
| *StaticArg* + *StaticArg*
| *StaticArg* · *StaticArg*
| *StaticArg* *StaticArg*
| *StaticArg* / *StaticArg*
| 1 / *StaticArg*
| *StaticArg* ^ *StaticArg*
| *StaticArg* per *StaticArg*
| *DUPreOp* *StaticArg*
| *StaticArg* *DUPostOp*
| NOT *StaticArg*
| *StaticArg* OR *StaticArg*
| *StaticArg* AND *StaticArg*
| *StaticArg* IMPLIES *StaticArg*
| *TypeRef*
| (*StaticArg*)
DUPreOp ::= *square* | *cubic* | *inverse*
DUPostOp ::= *squared* | *cubed*
ArraySize ::= *ExtentRange*(, *ExtentRange*)*
ExtentRange ::= *StaticArg*? # *StaticArg*?
| *StaticArg*? : *StaticArg*?
| *StaticArg*
Number ::= *IntLiteral*

G.18 Symbols and Operators

AssignOp ::= := | = | *Op* =
Accumulator ::= \sum | \prod | BIG *Op*

G.19 Identifiers

IdList ::= *Id*(, *Id*)*
Name ::= *Id*
| *opr* *Op*
DottedId ::= *Id*(. *Id*)*
BindId ::= *Id*
| -

Appendix H

Full Grammar for Fortress Implementors

In this chapter, we provide a complete definition of the Fortress concrete syntax. This syntax has been extracted from the parser-generator source files of an open source partial Fortress reference implementation [13], available at:

<http://fortress.sunsource.net>

The parser of this reference implementation is generated by Rats! [12], which generates “packrat parsers” that memoize intermediate results to ensure linear time performance in the presence of unlimited lookahead and backtracking. Rats! includes production naming, module modifications, and semantic predicates. Actions that generate semantic values are omitted for simplicity. See [12] for more information about the Rats! parser generator.

H.1 Components and APIs

```
File =
  CompilationUnit w
  / (w Imports)? w Exports (w Decls)? w
  / (w Imports)? w AbsDecls w
  / w Imports (w AbsDecls)? w

CompilationUnit =
  w Component w
  / w Api

Component = "component" w DottedId (w Imports)? w Exports (w Decls)? w "end"

Api = "api" w DottedId (w Imports)? (w AbsDecls)? w "end"

Imports = Import (wr Import)*

Import =
  "import" wr ImportFrom
  / "import" wr AliasedDottedIds

ImportFrom =
  "*" (wr "except" wr Names)? wr "from" wr DottedId
  / AliasedNames wr "from" wr DottedId

Names =
  Name
  / "{" w NameList w "}"
```



```

NameList = Name (w " ," w Name)*

AliasedNames =
  AliasedName
  / "{" w AliasedNameList w "}"

AliasedName =
  Id (wr "as" wr DottedId)?
  / "opr" w Op (w "as" w Op)?
  / "opr" w LeftEncloser w RightEncloser (w "as" w LeftEncloser w RightEncloser)?

AliasedNameList = AliasedName (w " ," w AliasedName)*

AliasedDottedIds =
  AliasedDottedId
  / "{" w AliasedDottedIdList w "}"

AliasedDottedId = DottedId (wr "as" wr DottedId)?

AliasedDottedIdList = AliasedDottedId (w " ," w AliasedDottedId)*

Exports = Export (wr Export)*

Export = "export" wr DottedIds

DottedIds =
  DottedId
  / "{" w DottedIdList w "}"

DottedIdList = DottedId (w " ," w DottedId)*

```

H.2 Top-level Declarations

```

Decls = Decl (br Decl)*

Decl =
  TraitDecl
  / ObjectDecl
  / VarDecl
  / FnDecl
  /* Not Yet Implemented *****
  / DimUnitDecl
  ***** */
  / TypeAlias
  / TestDecl
  / PropertyDecl
  / ExternalSyntax

AbsDecls = AbsDecl (br AbsDecl)*

AbsDecl =
  AbsTraitDecl
  / AbsObjectDecl
  / AbsVarDecl
  / AbsFnDecl
  /* Not Yet Implemented *****
  / DimUnitDecl
  ***** */
  / TypeAlias
  / TestDecl
  / PropertyDecl
  / AbsExternalSyntax

```

H.3 Trait and Object Declarations

```
TraitDecl = TraitHeader (w GoInATrait)? w "end"

TraitHeader = TraitMods? "trait" w Id (w StaticParams)? (w Extends)? (TraitClauses)?

/* Each trait clause cannot appear more than once. */
TraitClauses = (w TraitClause)+

TraitClause =
  w Excludes
  / w Comprises
  / w Where

GoInATrait =
  GoFrontInATrait (br GoBackInATrait)?
  / GoBackInATrait

GoFrontInATrait = GoesFrontInATrait (br GoesFrontInATrait)*

GoesFrontInATrait =
  AbsFldDecl
  / GetterSetterDecl
  / PropertyDecl

GoBackInATrait = GoesBackInATrait (br GoesBackInATrait)*

GoesBackInATrait =
  MdDecl
  / PropertyDecl

ObjectDecl = ObjectHeader (w GoInAnObject)? w "end"

ObjectHeader = ObjectMods? "object" w Id (w StaticParams)? (w ObjectValParam)?
              (w Extends)? FnClauses

ObjectValParam = "(" (w Params)? w ")"

Varargs := "transient" wr VarargsParam

Param :=
  FldMods? PlainParam
  / "transient" wr PlainParam

GoInAnObject =
  GoFrontInAnObject (br GoBackInAnObject)?
  / GoBackInAnObject

GoFrontInAnObject = GoesFrontInAnObject (br GoesFrontInAnObject)*

GoesFrontInAnObject =
  FldDecl
  / GetterSetterDef
  / PropertyDecl

GoBackInAnObject = GoesBackInAnObject (br GoesBackInAnObject)*

GoesBackInAnObject =
  MdDef
  / PropertyDecl

AbsTraitDecl = TraitHeader (w AbsGoInATrait)? w "end"

AbsGoInATrait =
  AbsGoFrontInATrait (br AbsGoBackInATrait)?
```

```

    / AbsGoBackInATrait
AbsGoFrontInATrait = AbsGoesFrontInATrait (br AbsGoesFrontInATrait)*

AbsGoesFrontInATrait =
    ApiFldDecl
    / AbsGetterSetterDecl
    / PropertyDecl

AbsGoBackInATrait = AbsGoesBackInATrait (br AbsGoesBackInATrait)*

AbsGoesBackInATrait =
    AbsMdDecl
    / AbsCoercion
    / PropertyDecl

AbsObjectDecl = ObjectHeader (w AbsGoInAnObject)? w "end"

AbsGoInAnObject =
    AbsGoFrontInAnObject (br AbsGoBackInAnObject)?
    / AbsGoBackInAnObject

AbsGoFrontInAnObject = AbsGoesFrontInAnObject (br AbsGoesFrontInAnObject)*

AbsGoesFrontInAnObject =
    ApiFldDecl
    / AbsGetterSetterDecl
    / PropertyDecl

AbsGoBackInAnObject = AbsGoesBackInAnObject (br AbsGoesBackInAnObject)*

AbsGoesBackInAnObject =
    AbsMdDecl
    / AbsCoercion
    / PropertyDecl

```

H.4 Variable Declarations

```

VarDecl =
    VarWTypes w InitVal
    / VarWoTypes w "=" w NoNewlineExpr
    / VarWoTypes w ":" w TypeRef w "..." w InitVal
    / VarWoTypes w ":" w SimpleTupleType w InitVal

VarWTypes =
    VarWType
    / "(" w VarWType (w "," w VarWType)+ w ")"

VarWType = VarMods? Id w IsType

VarWoTypes =
    VarWoType
    / "(" w VarWoType (w "," w VarWoType)+ w ")"

VarWoType = VarMods? Id

InitVal = ("=" / ":=") w NoNewlineExpr

AbsVarDecl =
    VarWTypes
    / VarWoTypes w ":" w TypeRef w "..."
    / VarWoTypes w ":" w SimpleTupleType

```

H.5 Function Declarations

```
FnDecl =
  FnDef
  / AbsFnDecl

FnDef = FnMods? FnHeaderFront FnHeaderClause w "=" w NoNewlineExpr

AbsFnDecl =
  FnMods? FnHeaderFront FnHeaderClause
  / Name w ":" w ArrowType

FnHeaderFront =
  Id (w StaticParams)? w ValParam
  / OpHeaderFront
```

H.6 Dimension, Unit, Type Alias, Test, Property, and External Syntax Declarations

```
/* Not Yet Implemented *****
DimUnitDecl =
  "dim" w Id (w "=" w DimRef)? (w "default" w Id)?
  / ("unit" / "SI_unit") w Id (wr Id)* (w ":" w DimRef)? (w "=" w NoNewlineExpr)?
  / "dim" w Id (w "=" w DimRef)? s ("unit" / "SI_unit") w Id (wr Id)* (w "=" w NoNewlineExpr)?
***** */

TypeAlias = "type" w Id (w StaticParams)? w "=" w TypeRef

TestDecl = "test" w Id w "[" w GeneratorList w "]" w "=" w NoNewlineExpr

PropertyDecl = "property" (w Id w "=")? (w "FORALL" w ValParam)? w NoNewlineExpr

ExternalSyntax = "syntax" w OpenExpander w Id w CloseExpander w "=" w NoNewlineExpr

OpenExpander =
  Id
  / LeftEncloser

CloseExpander =
  Id
  / RightEncloser
  / "end"

AbsExternalSyntax = "syntax" w OpenExpander w Id w CloseExpander
```

H.7 Headers

```
Extends = "extends" w TraitTypes

Excludes = "excludes" w TraitTypes

Comprises = "comprises" w ComprisingTypes

TraitTypes =
  TraitType
  / "{" w TraitTypeList w "}"

TraitTypeList = TraitType (w "," w TraitType)*
```

```

ComprisingTypes =
  TraitType
  / "{" w ComprisingTypeList w "}"

ComprisingTypeList =
  "..."
  / TraitType (w "," w TraitType)* (w "," w "...")?

Where = "where" w "{" w WhereClauseList w "}"

WhereClauseList = WhereClause (w "," w WhereClause)*

WhereClause =
  Id w Extends
  / TypeAlias
  / NatConstraint
  / IntConstraint
  / BoolConstraint
  / UnitConstraint
/* Not Yet Implemented *****
  / TypeRef w "coerces" w TypeRef
  / TypeRef w "widens" w TypeRef
***** */

NatConstraint =
  "nat" w Id
/* Not Yet Implemented *****
  / NatRef w "LT" w NatRef
  / NatRef w "LE" w NatRef
  / NatRef w "GT" w NatRef
  / NatRef w "GE" w NatRef
  / NatRef w "=" w NatRef
***** */
  / "(" w NatConstraint w ")"

/* Not Yet Implemented *****
NatRef = StaticArg
***** */

IntConstraint =
  "int" w Id
/* Not Yet Implemented *****
  / NatRef w "LT" w NatRef
  / NatRef w "LE" w NatRef
  / NatRef w "GT" w NatRef
  / NatRef w "GE" w NatRef
  / NatRef w "=" w NatRef
***** */
  / "(" w IntConstraint w ")"

BoolConstraint =
  "bool" w Id
/* Not Yet Implemented *****
  / "NOT" w BoolRef
  / BoolRef w "OR" w BoolRef
  / BoolRef w "AND" w BoolRef
  / BoolRef w "IMPLIES" w BoolRef
  / BoolRef w "=" w BoolRef
***** */
  / "(" w BoolConstraint w ")"

/* Not Yet Implemented *****
BoolRef = StaticArg
***** */

```

```

UnitConstraint =
  "unit" w Id
/* Not Yet Implemented *****
 / UnitRef w "=" w UnitRef
***** */

FnHeaderClause = (w IsType)? FnClauses

FnClauses = (w Throws)? (w Where)? Contract

Throws = "throws" w MayTraitTypes

MayTraitTypes =
  "{" w "}"
 / TraitTypes

Contract = (w Requires)? (w Ensures)? (w Invariant)?

Requires = "requires" w "{" (w ExprList)? w "}"

Ensures = "ensures" w "{" (w ProvidedList)? w "}"

ProvidedList = Provided (w "," w Provided)*

Provided = Expr (w "provided" w Expr)?

Invariant = "invariant" w "{" (w ExprList)? w "}"

CoercionClauses = (w Throws)? (w CoercionWhere)? w Contract

CoercionWhere = "where" w "{" w CoercionWhereClauseList w "}"

CoercionWhereClauseList = CoercionWhereClause (w "," w CoercionWhereClause)*

CoercionWhereClause =
  WhereClause
 / TypeRef w "widens" w "or" w "coerces" w TypeRef

UniversalMod =
  "private"
 / "test"

TraitMod =
  "value"
 / UniversalMod

/* Each modifier cannot appear more than once. */
TraitMods = TraitMod wr (TraitMod wr)*

/* Each modifier cannot appear more than once. */
ObjectMods = TraitMods

FnMod =
  LocalFnMod
 / UniversalMod

/* Each modifier cannot appear more than once. */
FnMods = FnMod wr (FnMod wr)*

VarMod =
  "var"
 / UniversalMod

/* Each modifier cannot appear more than once. */
VarMods = VarMod wr (VarMod wr)*

```

```

AbsFldMod =
  "hidden"
  / "settable"
  / "wrapped"
  / UniversalMod

/* Each modifier cannot appear more than once. */
AbsFldMods = AbsFldMod wr (AbsFldMod wr)*

FldMod =
  "var"
  / AbsFldMod

/* Each modifier cannot appear more than once. */
FldMods = FldMod wr (FldMod wr)*

ApiFldMod =
  "hidden"
  / "settable"
  / UniversalMod

/* Each modifier cannot appear more than once. */
ApiFldMods = ApiFldMod wr (ApiFldMod wr)*

LocalFnMod =
  "atomic"
  / "io"

/* Each modifier cannot appear more than once. */
LocalFnMods = LocalFnMod wr (LocalFnMod wr)*

StaticParams = "[\" w StaticParamList w \"]"

StaticParamList = StaticParam (w "," w StaticParam)*

StaticParam =
  Id (w Extends)? (w "absorbs" w "unit")?
  / "nat" w Id
  / "int" w Id
  / "bool" w Id
  / "dim" w Id
/* Not Yet Implemented *****
  / "unit" w Id (w ":" w DimRef)? (w "absorbs" w "unit")?
***** */
  / "opr" w Op
  / "ident" w Id

```

H.8 Parameters

```

ValParam =
  BindId
  / "(" (w Params)? w ")"

Params =
  (Param w "," w)* (Varargs w "," w)? Keyword (w "," w Keyword)*
  / (Param w "," w)* Varargs
  / Param (w "," w Param)*

VarargsParam = Id w ":" w TypeRef w "...

Varargs = VarargsParam

```

```

Keyword = Param w "=" w Expr

PlainParam =
  BindId (w IsType)?
  / TypeRef

Param = PlainParam

OpHeaderFront =
  "opr" (w StaticParams)? w (LeftEncloser / Encloser) w Params w
  (RightEncloser / Encloser) (w ":@" w "(" w SubscriptAssignParam w ")" )?
  / "opr" (w StaticParams)? w ValParam w Op
  / "opr" w (Op / Encloser) (w StaticParams)? w ValParam

SubscriptAssignParam =
  Varargs
  / Param

```

H.9 Method Declarations

```

MdDecl =
  MdDef
  / AbsMdDecl

MdDef =
  FnMods? MdHeaderFront FnHeaderClause w "=" w NoNewlineExpr
  / Coercion

AbsMdDecl = ("abstract" wr)? FnMods? MdHeaderFront FnHeaderClause

MdHeaderFront =
  (Receiver ".")? Id (w StaticParams)? w ValParam
  / OpHeaderFront

Receiver =
  Id
  / "self"

GetterSetterDecl =
  GetterSetterDef
  / AbsGetterSetterDecl

GetterSetterDef = FnMods? GetterSetterMod MdHeaderFront FnHeaderClause w "=" w NoNewlineExpr

GetterSetterMod =
  "getter" wr
  / "setter" wr

AbsGetterSetterDecl = (abstract wr)? FnMods? GetterSetterMod MdHeaderFront FnHeaderClause

Coercion = ("widening" w)? "coercion" (w StaticParams)? w "(" w Id w IsType w ")"
  CoercionClauses w "=" w NoNewlineExpr

AbsCoercion = ("widening" w)? "coercion" (w StaticParams)? w "(" w Id w IsType w ")"
  CoercionClauses

```

H.10 Method Parameters

```

ValParam := "(" (w Params)? w ")"

```



```

Param :=
  PlainParam
  / "self"

```

H.11 Field Declarations

```

FldDecl = VarDecl

VarWType := FldMods? Id w IsType

VarWoType := FldMods? Id

```

H.12 Abstract Field Declarations

```

AbsFldDecl = AbsVarDecl

VarWType := AbsFldMods? Id w IsType

VarWoType := AbsFldMods? Id

ApiFldDecl = ApiFldMods? Id w IsType

```

H.13 Expressions

```

/* Expr and UnitExpr are mutually left recursive.
   We've inlined them into each other and removed the left recursion. */
Expr = ExprFront ExprTail*

```

```

Expr ExprFront =
  <Assign> AssignLefts w AssignOp w Expr
  / OpExpr
  / DelimitedExpr
  / <Flow> FlowExpr
  / <Fn> "fn" w ValParam (w IsType)? (w Throws)? w "=" w Expr
/* Not Yet Implemented *****
  / UnitRef
***** */

```

```

ExprTail =
  <As> As
  / <Asif> AsIf
/* Not Yet Implemented *****
  / w UnitRef
  / w "DOT" w UnitRef
  / w "/" w UnitRef
  / w "per" w UnitRef
  / w "in" w UnitRef
***** */

```

```

As = w "as" w TypeRef

```

```

AsIf = w "asif" w TypeRef

```

```

/* Not Yet Implemented *****
UnitExpr =
  UnitRef
  / Expr w UnitRef
  / Expr w "DOT" w UnitRef

```

```

    / Expr w "/" w UnitRef
    / Expr w "per" w UnitRef
    / Expr w "in" w UnitRef

UnitRef = StaticArg
***** */

AssignLefts =
    "(" w AssignLeft (w "," w AssignLeft)* w ")"
    / AssignLeft

AssignLeft =
    PrimaryFront AssignLeftTail*
    / BindId

AssignLeftTail =
    SubscriptAssign
    / FieldSelectionAssign

SubscriptAssign = "[" (w ExprList)? w "]"

FieldSelectionAssign = "." Id

OpExpr =
    OpExprNoEnc
    / OpExprLeftEncloser
    / Encloser

OpExprNoEnc =
    OpExprPrimary
    / OpExprPrefix
    / Op

TightInfixRight =
    Encloser OpExprPrimary
    / Encloser OpExprPrefix
    / <Primary> Encloser wr OpExprPrimary
    / <Loose> Encloser wr LooseInfix
    / <LeftLoose> Encloser wr LeftLooseInfix
    / Encloser

LeftLooseInfix =
    OpExprLeftEncloser
    / <Primary> Encloser wr OpExprPrimary
    / <Prefix> Encloser wr OpExprPrefix
    / <Left> Encloser wr OpExprLeftEncloser

OpExprLeftEncloser = Encloser OpExprNoEnc

OpExprPrimary =
    Primary TightInfixPostfix
    / Primary TightInfixRight
    / <Primary> Primary wr OpExprPrimary
    / <Loose> Primary wr LooseInfix
    / <LeftLoose> Primary wr LeftLooseInfix
    / Primary

OpExprPrefix =
    Op OpExprPrimary
    / Op OpExprPrefix
    / Op OpExprLeftEncloser
    / <Primary> Op wr OpExprPrimary
    / <Prefix> Op wr OpExprPrefix
    / <Left> Op wr OpExprLeftEncloser

```

```

TightInfixPostfix =
    Op OpExprPrimary
  / Op OpExprPrefix
  / Op OpExprLeftEncloser
  / <Primary> Op wr OpExprPrimary
  / <Prefix> Op wr OpExprPrefix
  / <Left> Op wr OpExprLeftEncloser
  / Op

LooseInfix =
    Op wr OpExprPrimary
  / Op wr OpExprPrefix
  / <Left> Op wr OpExprLeftEncloser

Primary = PrimaryFront PrimaryTail*

PrimaryFront =
    Comprehension
  / Id "[" w StaticArgList w "\""
  / BaseExpr
  / LeftEncloser (w ExprList)? w RightEncloser
/* Not Yet Implemented *****
  / TraitType ".coercion" ("[" w StaticArgList w "\"")? "(" w Expr w ")"
***** */

PrimaryTail =
    SubscriptExpr
  / MethodInvocation
  / FieldSelection
  / ExponentExpr
  / TightJuxtaposition

SubscriptExpr = "[" (w ExprList)? w "]"

MethodInvocation = /* REVERSE ORDER */
    "." Id ("[" w StaticArgList w "\"")? TupleExpr
  / "." Id ("[" w StaticArgList w "\"")? "(" w ")"

FieldSelection = "." Id

ExponentExpr =
    "^" BaseExpr
  / ExponentOp

TightJuxtaposition = /* REVERSE ORDER */
    TupleExpr
  / "(" w ")"
  / Primary

FlowExpr =
    "exit" (w Id)? (w "with" w Expr)?
  / Accumulator (w "[" w GeneratorList w "]" )? w Expr
  / "atomic" w AtomicBack
  / "tryatomic" w AtomicBack
  / "spawn" w Expr
  / "throw" w Expr

AtomicBack =
    AssignLefts w AssignOp w Expr
  / OpExpr
  / DelimitedExpr

GeneratorList = Generator (w "," w Generator)*

Generator =

```

```

    Id w "<-" w Expr
  / "(" w Id w "," w IdList w ")" w "<-" w Expr
  / Expr

```

H.14 Expressions Enclosed by Keywords

```

DelimitedExpr =
  TupleExpr
  / "object" (w Extends)? (w GoInAnObject)? w "end"
  / Do
  / "label" w Id w BlockElems w "end" w Id
  / "while" w Expr w Do
  / "for" w GeneratorList w DoFront w "end"
  / "if" w Expr w "then" w BlockElems (w Elifs)? (w Else)? w "end"
  / "(" w "if" w Expr w "then" w BlockElems (w Elifs)? w Else (w "end")? w ")"
  / "case" w Expr (w Op)? w "of" w CaseClauses (w CaseElse)? w "end"
  / "case" w ("largest" / "smallest") (w Op)? w "of" w CaseClauses w "end"
  / "typecase" w TypecaseBindings w "of" w TypecaseClauses (br CaseElse)? w "end"
  / "try" w BlockElems (w Catch)? (w "forbid" w TraitTypes)? (w "finally" w BlockElems)? w "end"

Do = DoFront w "end"
/* Not Yet Implemented *****
(DoFront w "also" w)* DoFront w "end"
***** */

DoFront = "do" (w BlockElems)?
/* Not Yet Implemented *****
("at" w Expr w)? ("atomic" w)? "do" (w BlockElems)?
***** */

TupleExpr =
  "(" w (Expr w "," w)* (Expr w "..." w "," w)? Binding (w "," w Binding)* w ")"
  / NoKeyTuple

NoKeyTuple =
  "(" w (Expr w "," w)* Expr w "..." w ")"
  / "(" w (Expr w "," w)* Expr w ")"

Elifs = Elif (w Elif)*

Elif = "elif" w Expr w "then" w BlockElems

Else = "else" w BlockElems

CaseClauses = CaseClause (br CaseClause)*

/* CaseClause is defined in LocalDecl.rats */

CaseElse = "else" w match w BlockElems

TypecaseBindings =
  "(" w BindingList w ")"
  / Binding
  / Id

BindingList = Binding (w "," w Binding)*

Binding = BindId w "=" w Expr

TypecaseClauses = TypecaseClause (br TypecaseClause)*

TypecaseClause = TypecaseTypeRefs w match w BlockElems

```

```

TypecaseTypeRefs =
  "(" w TypeRefList w ")"
  / TypeRef

Catch = "catch" w Id w CatchClauses

CatchClauses = CatchClause (br CatchClause)*

CatchClause = TraitType w match w BlockElems

Comprehension =
  "{" w Expr wr bar wr GeneratorList w "}"
  / "{" w Entry wr bar wr GeneratorList w "}"
  / "<|" w Expr wr bar wr GeneratorList w "|>"
  / "[" w ArrayComprehensionClause (br ArrayComprehensionClause)* w "]"

/* The operator "|->" should not be in the left-hand sides of map expressions
and map/array comprehensions.
*/
mapstoOp = !("<|->" w Expr (w mapsto / wr bar)) "|->"

Entry = Expr w mapsto w Expr

ArrayComprehensionLeft =
  IdOrInt w mapsto w Expr
  / "(" w IdOrInt w "," w IdOrIntList w ")" w mapsto w Expr

/* ArrayComprehensionClause is defined in Symbol.rats */

IdOrInt =
  Id
  / IntLiteral

IdOrIntList = IdOrInt (w "," w IdOrInt)*

BaseExpr =
  NoKeyTuple
  / Literal
  / Id
  / "self"

ExprList = Expr (w "," w Expr)*

```

H.15 Local Declarations

```

BlockElems =
  BlockElem br BlockElems
  / BlockElem
  &(w Elifs / w Else / br CaseClause / br TypecaseTypeRefs / br CaseElse
  / w "end" / w Catch / w "forbid" / w "finally")
  / BlockElem w ";"
  &(w Elifs / w Else / w CaseClause / w TypecaseTypeRefs / w CaseElse
  / w "end" / w Catch / w "forbid" / w "finally")

BlockElem =
  LocalVarFnDecl
  / NoNewlineExpr (s " ," w NoNewlineGeneratorList)?

LocalVarFnDecl =
  LocalFnDecl (br LocalFnDecl)*
  / LocalVarDecl

LocalFnDecl = LocalFnMods? FnHeaderFront FnHeaderClause w "=" w NoNewlineExpr

```

```

VarDecl :=
  VarWTypes s InitVal
  / VarWTypes
  / VarWoTypes s "=" s NoNewlineExpr
  / VarWoTypes s ":" s TypeRef s "..." (s InitVal)?
  / VarWoTypes s ":" s SimpleTupleType (s InitVal)?
LocalVarDecl = VarDecl

VarWType := ("var" wr)? Id s ":" s TypeRef

VarWoType :=
  ("var" wr)? Id
  / Unpasting

Unpasting = "[" w UnpastingElems w "]"

UnpastingElems =
  UnpastingElem RectSeparator UnpastingElems
  / UnpastingElem

UnpastingElem =
  Id ("[" w UnpastingDim w "]")?
  / Unpasting

UnpastingDim = ExtentRange (w "BY" w ExtentRange)+

CaseClause = NoNewlineExpr w match w BlockElems

```

H.16 Literals

```

Literal =
  "(" w ")"
  / NumericLiteral
  / CharLiteral
  / StringLiteral
  / "{" w Entry (w "," w Entry)* w "}"
  / "[" w RectElements w "]"

RectElements = NoSpaceExpr MultiDimCons*

MultiDimCons = RectSeparator NoSpaceExpr

/* RectSeparator is defined in Spacing.rats */

NumericLiteral =
  FloatLiteral
  / IntLiteral

FloatLiteral = DigitString "." DigitString

IntLiteral = DigitString

DigitString = [0-9]+

CharLiteral = "'" CharLiteralContent "'"

StringLiteral = ["] StringLiteralContent* ["]
StringLiteralContent =
  EscapeSequence
  / (!["\\] _)
EscapeSequence = '\\\ [btnfr"'\]

```

```
CharLiteralContent = (!'"' _)
```

H.17 Expressions without Newlines

```
NoNewlineExpr = Expr
Expr := ...
ExprFront :=
  <Assign> AssignLefts s AssignOp w NoNewlineExpr
  / ...

ExprTail :=
  <As> NoNewlineAs
  / <AsIf> NoNewlineAsIf
/* *****
* Not Yet Implemented *****
/ s      UnitRef
/ s "DOT" w UnitRef
/ s "/" w UnitRef
/ s "per" w UnitRef
/ s "in" w UnitRef
***** */

NoNewlineAs = s "as" w TypeRef
NoNewlineAsIf = s "asif" w TypeRef

TightInfixRight := ...
  / <Primary>  Encloser sr OpExprPrimary
  / <Loose>    Encloser sr LooseInfix
  / <LeftLoose> Encloser sr LeftLooseInfix

LeftLooseInfix := ...
  / <Primary> Encloser sr OpExprPrimary
  / <Prefix> Encloser sr OpExprPrefix
  / <Left>   Encloser sr OpExprLeftEncloser

OpExprPrimary := ...
  / <Primary> Primary sr OpExprPrimary
  / <Loose>   Primary sr LooseInfix
  / <LeftLoose> Primary sr LeftLooseInfix

OpExprPrefix := ...
  / <Primary> Op sr OpExprPrimary
  / <Prefix> Op sr OpExprPrefix
  / <Left>   Op sr OpExprLeftEncloser

TightInfixPostfix := ...
  / <Primary> Op sr OpExprPrimary
  / <Prefix> Op sr OpExprPrefix
  / <Left>   Op sr OpExprLeftEncloser

LooseInfix := ...
  / <Left> Op sr OpExprLeftEncloser

GeneratorList := Generator (s ", " w Generator)*

NoNewlineGeneratorList = GeneratorList
```

H.18 Expressions within Array Expressions

```
ExprFront -= <Flow> , <Fn>
```

```

NoSpaceExpr = ExprFront

TightInfixRight == <Primary>, <Loose>, <LeftLoose>

OpExprPrimary == <Primary>, <Loose>, <LeftLoose>

OpExprPrefix == <Primary>, <Prefix>, <Left>

TightInfixPostfix == <Primary>, <Prefix>, <Left>

```

H.19 Types

```

IsType = ":" w TypeRef

/* TypeRef and TraitType/ArrowType/DimType are mutually left recursive.
   We've inlined them into each other and removed the left recursion. */
TypeRef =
  TraitType
/* Not Yet Implemented *****
 / DimType
***** */
 / TypeRefFront TypeRefTail*

TypeRefFront =
  TupleType
 / "(" (w TypeRef)? w ")"

TraitType = TraitTypeFront TypeRefTail*

TraitTypeFront =
  DottedId (w "[" w StaticArgList w "]" )?
 / "{" w TypeRef w mapsto w TypeRef w "}"
 / "<" w TypeRef w ">"

TypeRefTail =
  ArrayTypeTail
 / MatrixTypeTail
 / ArrowTypeTail
/* Not Yet Implemented *****
 / sr      DimRef
 / w "DOT" w DimRef
 / w "/"   w DimRef
 / w "per" w DimRef
 / sr      UnitRef
 / w "DOT" w UnitRef
 / w "/"   w UnitRef
 / w "per" w UnitRef
 / w "in"  w DimRef
***** */

ArrayTypeTail = w "[" (w ArraySize)? w "]"

MatrixTypeTail =
  "^" StaticArgFront
 / "^" "(" w ExtentRange (w "BY" w ExtentRange)* w ")"

ArrowTypeTail = w "->" w TypeRef (w Throws)?

ArrowType = (TraitTypeFront / TypeRefFront) ArrowTypeActionTail*

ArrowTypeActionTail = ArrowTypeAction
ArrowTypeAction = w "->" w TypeRef (w Throws)?

```



```

TupleType =
  "(" w (TypeRef w "," w)* (TypeRef w ".." w "," w)? KeywordType (w "," w KeywordType)* w ")"
  / "(" w (TypeRef w "," w)* TypeRef w ".." w ")"
  / SimpleTupleType

KeywordType = Id w "=" w TypeRef

SimpleTupleType = "(" w TypeRef w "," w TypeRefList w ")"

TypeRefList = TypeRef (w "," w TypeRef)*

/* Not Yet Implemented *****
DimType = DimTypeFront TypeRefTail*

DimTypeFront = DimRef

DimRef = StaticArg
***** */

StaticArgList = StaticArg (w "," w StaticArg)*

StaticArg = StaticArgFront StaticArgTail*

StaticArgFront =
  Number
  / Op
/* Not Yet Implemented *****
  / "true"
  / "false"
  / "Unity"
  / "dimensionless"
***** */
  / "1" w "/" w StaticArg
/* Not Yet Implemented *****
  / DUPreOp sr StaticArg
  / "NOT":op w StaticArg
***** */
  / TypeRef
  / "(" StaticArg ")"

StaticArgTail =
  SumStaticArg
  / ProductStaticArg
  / QuotientStaticArg
  / ExponentStaticArg
/* Not Yet Implemented *****
  / w "per" w StaticArg
  / w DUPostOp
  / w "OR":op w StaticArg
  / w "AND":op w StaticArg
  / w "IMPLIES":op w StaticArg
***** */

SumStaticArg = w "+" w StaticArg

ProductStaticArg = (w "DOT" w / sr) StaticArg

QuotientStaticArg = w "/" w StaticArg

ExponentStaticArg = "^" StaticArg

/* Not Yet Implemented *****
DUPreOp =
  "square"
  / "cubic"

```

```

    / "inverse"
    ***** */

/* Not Yet Implemented *****
DUPostOp =
    squared
    / cubed
    ***** */

ArraySize = ExtentRange (w "," w ExtentRange)*

ExtentRange =
    (StaticArg w)? "#" (w StaticArg)?
/* Not Yet Implemented *****
    / (StaticArg w)? ":" (w StaticArg)?
    ***** */
    / StaticArg

Number = IntLiteral

```

H.20 Symbols and Operators

```

Encloser = encloser

LeftEncloser = leftEncloser

RightEncloser = rightEncloser

ExponentOp = exponentOp

bar = &("|" wr GeneratorList closingComprehension) "|"
closingComprehension =
    w "]"
    / w "|>"
    / br ArrayComprehensionClause
    / w "]"
sd = [*.]?
bars = "|" (sd "|")*
slashes = "/" (sd "/" )*
    / "\\" (sd "\\")*
lesses = "<" (sd "<")*
greateres = ">" (sd ">")*

encloser =
    "///" / "/"
    / !(bar) bars !([*.>/\\] / "->")

leftEncloser =
    leftEncloserMulti
    / !('|') _

leftEncloserMulti =
    "(" ("/"+ / "\\")+
    / "[" (sd slashes)
    / "{" (sd slashes)
    / lesses sd (slashes / bars)
    / bars sd slashes
    / "{"* / "["*

rightEncloser =
    rightEncloserMulti
    / !('|') _

```

```

rightEncloserMulti =
    "/"+" )"
    / "\\ "+" )"
    / slashes sd (greater / bars / [\\])
    / bars sd greater
    / "*" / "}"
    / "]" / "}"

exponentOp = "^T" / "^"

OpName = !(accumulator / "BY") id

Op = compOp / op !(equalsOp)

compOp =
    "=="
    / "=/"
    / "<="
    / ">="

multiOp =
    "-/->"
    / "<-/-"
    / "-->"
    / "=="
    / ">>>"
    / mapstoOp
    / "<<<"
    / "<->"
    / "<="
    / "->"
    / doublerightarrow
    / ">>"
    / "<<"
    / "*"
    / "!!"

singleOp = !(encloser / leftEncloser / rightEncloser / multiOp / compOp / match) _

op = OpName
    / multiOp
    / singleOp
    / exponentOp

CompoundOp = op equalsOp

/* The operator "=>" should not be in the left-hand sides of case/typecase expressions. */
doublerightarrow = "=>" &(w NoNewlineExpr w match)
match = "=>"

/* The operator "BY" should not be used with ExtentRange. */
crossOp = "BY":op &(w ExtentRange)

equalsOp = "=":singleOp

AssignOp =
    ":@"
    / CompoundOp

accumulator = "SUM" / "PROD"

Accumulator =
    "SUM"
    / "PROD"
    / "BIG" w Op

```

```
ArrayComprehensionClause = ArrayComprehensionLeft wr bar wr GeneratorList
```

H.21 Identifiers

```
id      = idstart idrest*
idstart = UnicodeIdStart / _
idrest  = UnicodeIdStart / "'" / UnicodeIdRest
IdName  = id

Id = IdName

IdList = Id (w ", " w Id)*

Name =
    Id
    / "opr" w Op

DottedId = /* If we find ..., the dot doesn't count */
    Id &(w "...")
    / Id ( "." Id)*

BindId =
    Id
    / _
```

H.22 Spaces and Comments

```
Whitespace = Space / Newline
Space      = ' ' / '\t' / '\f' / Comment
Newline   = '\r' '\n' / '\r' / '\n'
Comment   = "(*" (w CommentContent)* w "*)"
CommentContent = Comment / ('*' '!')' / !'*' _

w = wr?
wr = Whitespace w

s = sr?
sr = Space s

nl = s Newline w
br = nl / s ";" w

RectSeparator =
    w ";" w
    / sr
    / nl
```

Appendix I

Changes between Fortress 1.0 α and 1.0 β Specifications

- The Fortress type hierarchy has changed: `Any` is the single root of type hierarchy. The immediate subtypes of `Any` are `Tuple`, `()`, and `Object`. Some library code has changed due to the change of type hierarchy.
- Numeric literals may include narrow spaces.
- Elaborated stories:
 - Execution of a program including top-level initialization (Chapter 4)
 - Reduction variables (Section 4.4)
 - Lexical structure (Chapter 5)
 - Declarations (Chapter 6)
 - Names (Chapter 7)
 - Field accesses and field initialization order for objects (Section 10.2).
 - Operator and identifier parameters (Section 11.5)
 - Evaluation order of each expression (Chapter 13)
 - Spawn expressions (Section 13.23)
 - Static expressions (Section 13.26)
 - Overloading (Chapter 15 and Chapter 33)
 - Coercion (Chapter 17)
 - Support for Unicode input in ASCII (Appendix E)
- Libraries have changed:
 - The trait `Tuple` has been added. (Section 23.3)
 - Methods are added to the trait `Boolean`. (Section 24.1)
 - Basic integers have more properties. (Section 25.2)
 - The trait `Thread` is parameterized with a static type of the expression spawned. (Chapter 28)

- Traits describing algebraic constraints have been revised. Only the traits `HasLeftZeroes` and `HasRightZeroes` have identifier parameters. More traits have been added. (Chapter 37)
- Some syntax has changed:
 - Concrete syntax grammar is written in Extended BNF using ‘?’ for optional things instead of enclosing square brackets.
 - Grammars in the specification have been updated to the grammar implemented in the Fortress interpreter (but not all language features have been implemented). Here are the syntax changes:
 - * A file may contain a single component or API. The enclosing component or API declaration may be omitted.
 - * At least one export statement is required for a component.
 - * Top-level variable declarations and field declarations should have initial-value expressions.
 - * A single declaration may declare multiple top-level variables, local variables, or fields depending on the context. Immutable variables cannot be declared using the “:=” token.
 - * A default unit for a dimension must be an identifier.
 - * No abstract declarations of dimensions, units, and type aliases.
 - * Trait headers can have the `excludes`, `comprises`, and `where` clauses in any order.
 - * No empty `comprises` clauses. A `comprises` clause of a trait *T* may include “...” to hide some of *T*’s subtraits. A type listed in the `comprises` clause must be defined within the same component (or possibly an API imported by the component).
 - * More `where`-clause constraints and `bool` static arguments are added.
 - * Each contract clause (`requires` clause, `ensures` clause, and `invariant` clause) requires its sub-clauses or subexpressions to be separated by commas and enclosed by curly braces.
 - * Each modifier must not appear multiple times.
 - * Method declarations other than `getter` and `setter` declarations occur syntactically after field declarations and `getter` and `setter` declarations. Property declarations can be freely commingled with the field and method declarations. A `getter` or `setter` modifier should be the last modifier of a method declaration.
 - * Local function declarations have the same syntax as top-level function declarations except that local function declarations must not have the modifiers “`private`” and “`test`”.
 - * The second value parameter of a subscripted assignment operator method declaration must contain exactly one non-keyword value parameter.
 - * Left-hand sides of assignment expressions are (possibly multiple of) array indexing, field accesses, the ‘_’ token, or identifiers.
 - * Object expressions may include property declarations.
 - * The `else` clause of `case` or `typecase` expressions has the ‘`⇒`’ token right after “`else`”.
 - * A `typecase` expression uses “`of`” instead of “`in`” after its bindings.
 - * A parallel block expression syntax has been enriched.
 - * The ‘_’ token can be used in any place where a binding can occur.

- * An index in left-hand sides of array comprehensions is either an identifier or an integer numeral.
 - * Matrix and vector types can be abbreviated using superscripts.
 - * Alternative mathematical notations for arrow types are not supported any more.
 - * *DimRef*, *UnitRef*, *NatRef*, *IntRef*, and *BoolRef* are merged with *StaticArg*.
- More examples are added.

Bibliography

- [1] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hlzl, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. *The Self Programmer's Reference Manual*. http://research.sun.com/self/release_4.0/Self-4.0/manuals/Self-4.1-Pgmers-Ref.pdf, 2000.
- [2] Eric Allen, Victor Luchangco, and Sam Tobin-Hochstadt. Encapsulated Upgradable Components, March 2005.
- [3] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico.*, pages 356–368, November 1994.
- [4] Robert D. Blumofe, Christopher F. Joerg, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded run-time system. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 132–141, Montreal, Canada, 17–19 June 1998. ACM, SIGPLAN Notices.
- [5] Gilad Bracha, Guy Steele, Bill Joy, and James Gosling. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.
- [6] R. Cartwright and G. Steele. Compatible genericity with run-time types for the Java Programming Language. In *OOPSLA*, 1998.
- [7] William Clinger. Macros that work. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 155–162. ACM Press, 1991.
- [8] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006.
- [9] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in scheme. *Journal of LISP and Symbolic Computation*, 5(4):295–326, 1992.
- [10] Robert B. Findler, Mario Latendresse, and Matthias Felleisen. Behavioral contracts and behavioral subtyping. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 229–236. ACM Press, September 2001.
- [11] Seth C. Goldstein, Klaus E. Schauer, and Dave E. Culler. Lazy Threads: Implementing a Fast Parallel Call. *Journal of Parallel and Distributed Computing*, 37(1), August 1996.
- [12] Robert Grimm. Rats! – an easily extensible parser generator. <http://cs.nyu.edu/~rgrimm/xtc/rats.html>.
- [13] Sun's Programming Language Research Group. Project fortress. <http://fortress.sunsource.net>.
- [14] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '99)*, volume 34(10), pages 132–146, N. Y., 1999.

- [15] Richard Kelsey, William Clinger, and Jonathan Rees. Revised⁵ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.
- [16] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rmy, and Jérôme Vouillon. *The Objective Caml System, release 3.08*. <http://caml.inria.fr/distrib/ocaml-3.08/ocaml-3.08-refman.pdf>, 2004.
- [17] J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A Visual Environment for Developing Context-Sensitive Term Rewriting Systems (system description). In V. van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA-04*, LNCS 3091, pages 301–311, Valencia, Spain, June 3-5, 2004. Springer.
- [18] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.
- [19] Todd Millstein and Craig Chambers. Modular statically typed multimethods. *Information and Computation*, 175(1):76–118, May 2002.
- [20] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
- [21] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. Technical Report TM-449, MIT/LCS, 1991.
- [22] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., March 1966.
- [23] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. *The Scala Language Specification*. <http://scala.epfl.ch/docu/files/ScalaReference.pdf>, 2004.
- [24] OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface Version 2.0*. http://www.openmp.org/specs/mp-documents/fspec20_bars.pdf, November 2000.
- [25] Simon Peyton-Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.
- [26] Barry N. Taylor. Guide for the use of the international system of units (si). Technical report, United States Department of Commerce, National Institute of Standards and Technology, April 1995.
- [27] The Unicode Consortium. *The Unicode Standard, Version 4.0*. Addison-Wesley, 2003.
- [28] Java(TM) 2 Platform Standard Edition 6.0 API Specification. <http://download.java.net/jdk6/docs/api/index.html>.