University of California

Los Angeles

Enforcing and Validating User-Defined Programming Disciplines

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Shane Andrew Markstrum

2009

© Copyright by

Shane Andrew Markstrum

2009

The dissertation of Shane Andrew Markstrum is approved.

D. Stott Parker

Jens Palsberg

Sorin Lerner

Todd Millstein, Committee Chair

University of California, Los Angeles

2009

To Joe and Phyllis for always supporting my educational dreams

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Acknowledgments

I would first like to thank my wife, Tingting. She has shown me constant support and patience as I ran around the world doing research. She never lets me lose sight of my goals and always inspires me to do my best.

I am indebted to my advisor Todd Millstein for taking a chance on me when he came to UCLA. He has constantly challenged me to think creatively and to be rigorous in my work.

I want to thank my other committee members Sorin Lerner, Jens Palsberg, and Stott Parker for forcing me to find how my work fits into the broader spectrum of computer science research.

As collaborators, colleagues, and friends, I want to thank everyone in the TERTL and LASR labs at UCLA. I especially want to thank those unfortunate few who shared a cubicle with me and had to deal with my constant paper mess: Jeff Fisher, Nikitas Liogkas, Dan Marino, and Manav Mital. I also want to thank Ru-Gang Xu and Roy Shea for working with me on projects that did not prove fruitful in terms of papers, but were still enlightening.

I want to thank Brian Chin, Todd Millstein, and Jens Palsberg for their collaboration on the CLARITY project. I still think it would have been better to call it Disco Biscuit, though.

I want to thank Chris Andreae and James Noble at Victoria University of Wellington and Todd Millstein for their collaboration on JAVACOP. James especially went out of his way to put me up in New Zealand and introduced me to the Kiwi research experience: extreme distances and constant jetlag.

I am also indebted to Robert Fuhrer, Vijay Saraswat, and Frank Tip at IBM Research for the research opportunities they afforded me and their invaluable feedback

on CLARITY and JAVACOP.

Finally, I want to thank my parents for their love and support. I feel lucky to have parents that have encouraged and inspired my interest in teaching and were patient enough to let me go through the Ph.D. process at my own speed.

Vita

1980	Born, Orange, CA.
1994	First official language course–FORTRAN.
1996	Personal website published in Newsweek.
2002	Joint Bachelor of Science in Computer Science and Mathematics, Harvey Mudd College.
2002–2006	Substitute and summer school teacher in the Fullerton Joint Union High School District.
2003–2004	Teaching Assistant, Computer Science Department, UCLA.
2003–2009	Graduate Student Researcher, Computer Science Department, UCLA.
2004	Masters of Science in Computer Science, UCLA.
2006	Research Intern, IBM T.J. Watson Research Labs.
2008	Grant awarded for research in New Zealand, NSF East Asia and Pacific Summer Institute.

PUBLICATIONS

Kevin Eustice, Leonard Kleinrock, Shane Markstrum, Gerald Popek, V. Ramakrishna,

Peter Reiher. Enabling Secure Ubiquitous Interactions. In *Proceedings of the 1st International Workshop on Middleware for Pervasive and Ad-Hoc Computing*. Rio de Janiero, Brazil. June, 2003.

Kevin Eustice, Leonard Kleinrock, Shane Markstrum, Gerald Popek, V. Ramakrishna, Peter Reiher. Securing WiFi Nomads: The Case for Quarantine, Examination, and Decontamination. In *Proceedings of the New Security Paradigms Workshop 2003*. Ascona, Switzerland. August, 2003.

Everett Anderson, Kevin Eustice, Shane Markstrum, Mark Hansen, Peter Reiher. Mobile Contagion: Simulation of Infection and Disease. In *Proceedings of the IEEE Symposium on Measurement, Modeling, and Simulation of Malware*. Monterey, CA. June, 2005.

Brian Chin, Shane Markstrum, Todd Millstein. Semantic Type Qualifiers. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. Chicago, IL. June, 2005.

Brian Chin, Shane Markstrum, Todd Millstein, Jens Palsberg. Inference of User-Defined Type Qualifiers and Qualifier Rules. In *Proceedings of the 15th European Symposium on Programming*. Vienna, Austria. March, 2006.

Chris Andreae, James Noble, Shane Markstrum, Todd Millstein. A Framework for Implementing Pluggable Type Systems. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Portland, OR. October, 2006.

Brian Chin, Daniel Marino, Shane Markstrum, Todd Millstein. Enforcing and Validating User-Defined Programming Disciplines. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. San Diego, CA. June, 2007.

Shane Markstrum, Robert Fuhrer, Todd Millstein. Towards Concurrency Refactoring for X10. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Raleigh, NC. February, 2009

Abstract of the Dissertation

Enforcing and Validating User-Defined Programming Disciplines

by

Shane Andrew Markstrum

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2009

Professor Todd Millstein, Chair

Developing good software requires a large investment of time and money. Yet, production-quality code often has lurking bugs and security vulnerabilities. To help manage the complexity of building robust software systems, developers use *programming disciplines* during the creation process such as naming schemes for understandability, design patterns for extensibility, and lock ordering schemes to prevent program deadlock. But these disciplines are only informally specified and lack tooling support that would allow them to be enforced consistently on the code.

In this dissertation, I present a solution for creating frameworks for programming disciplines that allows programmers to define how disciplines should be enforced. Further, this solution allows users to validate that the disciplines they are using ensure desired program properties in their code. This solution is built upon three insights: a domain-specific language provides a standard way to specify a discipline; type systems provide a scaffolding for automatic discipline checking; and explicit association of a runtime invariant allows disciplines to be verified.

I present an overview of two instantiations of such programmer-defined discipline frameworks that I developed and built: JAVACOP for Java, and CLARITY for C. I show

how the use of these frameworks can be beneficial to programmers via case studies of disciplines–including design pattern checkers, untainted types, and non-null types–that find errors in real code. These frameworks further show a trade-off among the desired properties of expressiveness, usability, and reliability. JAVACOP is expressive in that the language can be used to write a wide variety of disciplines, but it cannot automatically validate that a discipline is sound with respect to its invariant. On the other hand, CLARITY has limited expressiveness but can automatically prove that a discipline establishes an invariant on a program.

Terminology

As my dissertation deals with metaprogramming–that is, programs that take other programs as input–it will help to define a few terms here that might otherwise be ambiguous to the reader.

base type system the type system built into a target language.

pluggable type system an optional type system or type system extension for a target language which may be provided as an argument to a type checker.

target language the programming language whose programs are meant to implement the discipline.

target program a program that is checked for proper use of a discipline.

type annotation a metadata tag that may be used to provide additional information about the type of a program construct but is generally ignored by the base type system. Also may be referred to as a program annotation.

type system extension a set of rules that extends an already existing type system without redefining the base type system. Also may be refereed to as a type refinement.

user generally, the programmer who is using disciplines. May occasionally refer to the person who is using software intended for the end-user.

user-defined programming discipline a programming discipline that the user has defined outside a target language. Also may be referred to as a user-defined discipline.

CHAPTER 1

Introduction

1.1 Programming Disciplines

Software developers today face a difficult task in building and maintaining efficient, reliable, and useful systems. To create good software, developers have to reason about a vast panoply of potential problems. Examples of these problems include making sure that null pointers are not dereferenced; being able to safely and successfully link dynamic software components like device drivers at runtime; and properly using locks to access shared data in a multi-threaded/concurrent runtime environment. Dealing with each of these problems individually is a difficult task; the difficulty is only compounded by having to handle all of them at once.

Current software best practices attempt to make this reasoning process manageable by using what I call *programming disciplines* during development. Programming disciplines are syntactic program constraints that ensure programs have certain desirable properties and meet certain design criteria. Examples of disciplines include *naming schemes* such as Hungarian notation and CamelCase which help to clarify the type and intent of program components; *non-null checks* placed before important pointer dereferences that help ensure accidental null pointer accesses do not cause a program to terminate unexpectedly; *design patterns* such as the `State` and `Visitor` patterns that provide ways to capture desired program behaviors and can simplify future program extension; and *locking schemes* which ensure that harmful data races are prevented in

a concurrent program.

Unfortunately, several problems currently limit the usefulness of programming disciplines. First, there is no standard way to specify how to enforce a discipline. Without such a standard, disciplines are limited to being defined informally in the comments of a program or in the project specification documents. Second, and related to the first problem, there is no way of automating the enforcement of a discipline on a target program. Current best practices establish that disciplines are enforced upon manual review of code. As manual enforcement of disciplines is imperfect, it is easy for violations of the disciplines to remain part of the code even after review. Thus, the reviewed programs may exhibit bugs and vulnerabilities when run. Third, there is no explicit way to state what program properties a discipline is meant to ensure. Without this ability, it is impossible to determine whether a discipline is really the correct discipline for ensuring a particular set of program properties.

In this dissertation, I present a solution that allows programmers to define how disciplines will be enforced and effectively automates the enforcement of programming disciplines on target programs. My solution ensures that intended bugs and vulnerabilities are reduced and/or prevented and helps verify that intended program properties are attained. This solution is scalable, as it supports the enforcement of any number of disciplines at the same time; non-intrusive, as it does not dramatically alter a language; and adaptable, in that programmers control how a disciplines should be enforced.

1.1.1 Discipline Examples

In this section, I present two examples of Java security vulnerabilities and explain how disciplines can be used to prevent the creation and release of code exhibiting these vulnerabilities. The first example focuses on SQL injection attacks: well-known, yet frequently-seen, security vulnerabilities for systems that interface with SQL databases.

The second example discusses a classic Java security vulnerability related to leaked private objects being manipulated in untrusted sections of code.

Both of these examples demonstrate that disciplines can be useful for preventing mistakes and bugs. With manual enforcement of the disciplines, though, it remains easy for the security vulnerabilities to elude detection.

SQL Injection Attacks SQL injection attacks [US 09] are a form of security vulnerability in programs that interact with SQL databases. In vulnerable programs, malicious users can inject new commands into program-generated queries. The malicious users accomplish this by providing unanticipated input which changes the nature of a query embedded in the program. Since SQL databased have seen wide-spread use in popular software like the Microsoft IIS web server, SQL injection attacks continue to be disruptive to e-commerce and Internet communication.

Listing 1.1 defines a simple Java API for interacting with a SQL database. In this API, the `executeQuery` method takes in a `String` which defines a SQL query and returns the results of performing the query on an associated database; and the `getName` method returns an unknown `String` from the user. Listing 1.2 shows a small code snippet that demonstrates how SQL injections can be introduced into a program. In this case, the query string `q` is generated by inserting the string returned from the `getName` method into a standard query template. If the returned string includes the `'` character then it will terminate the string embedded in the query and allow arbitrary new SQL operators to be included after that point. For example, if a user supplied the string `"';DROP users;--"`, then performing the query will result in the deletion of the `users` table from the database.

To prevent SQL injection attacks, developers can follow a very simple discipline: only *untainted* strings may be used to generate SQL queries. A tainted string is any

```
/* Executes a SQL query (given as a parameter) on the
 * currently affiliated SQL database and returns the
 * results.
 */
ResultSet executeQuery(String sql);

/* Returns a String provided by the user of this program.
 */
String getName();
```

Listing 1.1: A small Java API for programs that receive user input for querying SQL databases.

```
String name = o.getName();
String q = "SELECT * FROM users WHERE name='"+name+"';";
result = db.executeQuery(q);
```

Listing 1.2: A code snippet using the API from Listing 1.1 that demonstrates a possible SQL injection attack.

string that may have been manipulated or given as input from outside of the current scope. Untainted strings are then strings that are known to be safe to the program. Tainted strings in the case of SQL injection attacks can usually be made untainted through *sanitizing*: escaping errant `'` characters and removing disallowed operations from the strings. Listing 1.3 shows how this discipline can be applied to the code from Listing 1.2 to prevent the SQL injection attack. In this code snippet, the developer calls the `sanitize` method on the user input, thereby changing the input from a tainted string to an untainted one.

In a small excerpt of code, such as in Figures 1.2 and 1.3, it is easy to see how careful use of this discipline would result in code without SQL injection attacks. However, in the context of a large body of code, there can be many different paths that contribute to the generation of a query. Manually verifying that query component strings remain untainted on all paths to an invocation of `executeQuery` is a difficult task. Even

```
String name = o.getName();
String q = "SELECT * FROM users WHERE name='"
           +sanitize(name)+"';";
result = db.executeQuery(q);
```

Listing 1.3: A modified version of the code from Listing 1.2 which demonstrates proper use of an *untainted* discipline for SQL query components.

```
public class Class {
  ⋮
  private Identity[] signers;
  public Identity[] getSigners( ) { return signers; }
  ⋮
}
```

Listing 1.4: A snippet of the Java `Class` class from the Java Development Kit (JDK) 1.1 relating to class signers that reveals a security vulnerability.

though the discipline itself is simple to understand and communicate, it is nevertheless a non-trivial task to enforce.

Java Class Signing[1] In version 1.1 of Java, applets were able to gain access to certain privileged operations on a user's system by being signed by a trusted signer. Object signing was meant to prevent applets from causing damage to an applet user's system by requiring particular signatures in order to delete or modify files or access information. However, in the initial implementation of the `Class` class, there was a security vulnerability that allowed applets to bypass the signing method. The relevant excerpt of the JDK 1.1 `Class` definition is shown in Listing 1.4.

The problem is ultimately a result of the `getSigners` method returning an alias to the private `signers` field. A malicious applet could ask the Java security framework to give it the list of trusted signers for the system and add them to its own class's

[1] Code excerpts in Listings 1.4 and 1.6 are modified snippets of code from [VB99].

```
Enumeration trustedSigners =
  IdentityScope.getSystemScope().identities();
Identity[] mySigners = this.getClass().getSigners();
for(int i=0; i<mySigners.length
             && trustedSigners.hasMoreElements(); ++i){
  mySigners[i] = trustedSigners.nextElement();
}
⋮
/* Do bad things with new privileged signatures */
⋮
```

Listing 1.5: Code excerpt that illustrates how applets could exploit the JDK 1.1 signers vulnerability.

`signers` array through the alias returned from `getSigners`. An example of this is shown in Listing 1.5. As a result, when the applet attempted to perform privileged actions on the user's system, it would appear to be endorsed by the trusted signers.

A discipline for preventing this vulnerability can be stated as: *references to secure data should remain confined in the privileged scope of the defining package*. This *confined* discipline requires methods never return direct references to secure, private data to an unknown external scope. This restriction prevents outside tampering. Instead, a reference to a copy of the data can be returned or some form of immutable reference through which updates cannot occur. Applying the former option to the `Class` class results in the code excerpt in Listing 1.6.

The *confined* discipline [VB99] explains how to solve this security problem, but the discipline itself is not suitable for manual enforcement. Consider the case when a previously non-secure field is changed to a confined field. Then all prior existing references to that field are potential security holes. Manually checking all method calls and dereferences in a Java program for proper confined reference usage in any execution context is very tedious, especially for large programs. It is likely that such a

```
public class Class {
  ⋮
  private Identity[] signers;
  public Identity[] getSigners( ) {
    Identity[] pub = new Identity[signers.length];
    for (int i = 0; i < signers.length; i++)
      pub[i] = signers[i];
    return pub;
  }
  ⋮
}
```

Listing 1.6: A safe version of the `Class` class that returns a copy of the class's signers.

search would result in the improper classification of certain references.

1.2 Thesis Proposal: Enforcing and Validating User-Defined Programming Disciplines

In this dissertation, I present an approach to automatic enforcement of *user-defined* disciplines. This approach gives developers control over the disciplines they want to use on programs while still remaining scalable and adaptable. My approach is validated through the creation of two practical frameworks that target the languages of Java and C, respectively.

There are three key components that make user-defined discipline checking frameworks practical: a domain-specific language for specifying disciplines, discipline enforcement via type system extensions, and discipline verification against a run-time invariant.

1. **Discipline Specification**: The frameworks provide a domain-specific language (DSL) that allows programmers to easily write their own disciplines. This en-

ables developers to leverage a wide variety of disciplines when building their projects. Providing a language solves the issue of scalability as developers are not limited by a fixed set of built-in discipline checkers but can enforce the disciplines they want to enforce. It also solves the issue of adaptability since the users can modify a discipline specification to enforce the discipline in the manner they would like it to be enforced.

2. **Discipline Enforcement**: Static type systems already define and enforce a kind of discipline on programs and are already familiar to most programmers. Extending a type system to enforce more properties is a natural way to enforce new disciplines. Type systems have an advantage in checking disciplines since they are inherently modular and scalable. Thus, disciplines that are enforced as type system extensions can be used to incrementally check program components while still providing guarantees about the overall semantics of a program. Many programming languages also provide the ability to add metadata tags to types which provides a simple way to introduce new discipline information into a program.

3. **Discipline Validation**: Disciplines are used to ensure that programs exhibit certain desired program properties. These properties, in turn, can often be expressed as run-time invariants. An invariant can further be used to validate that the discipline actually establishes such properties. Specifying the invariant as a run-time invariant enables a wide variety of techniques to use in verifying the discipline including run-time testing, model checking, and proof assistance.

This definition of a discipline framework with the three components defined above leads to the thesis of this dissertation:

A framework for user-defined programming disciplines enables declarative speci-

fication and automated enforcement of many programming disciplines, both general-purpose and domain-specific, that find important errors and ensure desirable program properties.

I have further identified three characteristic properties of practical discipline frameworks. The properties are defined as follows:

1. **Expressiveness**: The framework is capable of expressing and enforcing many different disciplines. In particular, it handles disciplines that were outside of the set of discipline examples that initially motivated the framework's creation.

2. **Usability**: Developing discipline implementations in the framework feels familiar and/or natural to developers that use the target language. The framework provides straightforward tools that are easy to use and can be added to a standard toolchain. Discipline enforcement is scalable and adaptable.

3. **Verifiability**: Implementations enforce the program discipline which they are intended to enforce. The framework provides support for determining whether a discipline specification matches its design intent.

In practice, frameworks demonstrate a trade-off among these three properties. Each framework can effectively maximize two out of the three properties defined here at the expense of a reduction of the third property. In my implementations, I chose to emphasize usability and varied the focus on the other two properties to show how this trade-off works in practice. That the frameworks are usable is primarily driven by their emphasis on type systems.

1.2.1 Practical Implementations of My Approach

To support my thesis statement, I have created two practical user-defined discipline checking frameworks based on the approach described previously, written a variety of disciplines for these frameworks, and used them to find real bugs in existing software. These two frameworks represent two different points in the design space of discipline checking frameworks: one focuses on maximizing expressiveness and usability, and the other focuses on maximizing verifiability and usability.

The JAVACOP discipline framework for Java [ANM06, MME] focuses on expressiveness and usability. It provides a declarative rule and predicate language for specifying rules on Java programs, utilizing Java 1.5 metadata facilities for program annotation. I have implemented a number of interesting and non-trivial programming disciplines with it, including non-null types, confined types, design pattern checkers, and domain-specific checkers for Enterprise JavaBeans 3.0 and Safety Critical Java. Because of the focus on expressiveness and usability, verifiability of JAVACOP disciplines provides a challenge. However, to meet this challenge, the JAVACOP framework also incorporates a novel testing framework for practical validation of disciplines which utilizes runtime instrumentation for dynamic invariant checking. The JAVACOP framework was developed jointly with Chris Andreae and James Noble from Victoria University of Wellington, New Zealand, and Dan Marino and Todd Millstein from UCLA. JAVACOP is discussed further in Chapter 3.

The CLARITY discipline framework for C [CMM05a, CMM06] focuses on verifiability and usability. It provides a declarative rule language for specifying type qualifier rules that constrain the values of C expressions, including boolean, arithmetic, and pointer expressions. I was able to develop and validate a number of non-trivial integer type qualifiers, such as non-zero, positive, negative, and tainted/untainted. CLARITY also provides a soundness checker that uses an automated theorem prover to automati-

cally validate the qualifier specifications against a semantic invariant. This automated soundness checker would not be possible to reproduce for a system as expressive as JAVACOP. CLARITY was the first implementation based on this new user-defined discipline checking framework model. The CLARITY framework was developed jointly with Brian Chin, Todd Millstein, and Jens Palsberg at UCLA. A deeper exploration of CLARITY can be found in Chapter 4.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows.

Chapter 2 presents related work to this dissertation and discusses why this previous work does not solve the problems identified here.

Chapter 3 discusses JAVACOP, the pluggable type system framework for Java. In this chapter I provide an example-driven overview of its rule language, highlighting the constructs which stress the flexibility and expressiveness of the framework. I then provide insight into the semantics of the JAVACOP language by providing a translation from JAVACOP into Datalog$^{\neg}$. This is followed an exploration of the advanced features of JAVACOP that include a dataflow analysis framework and a pluggable type unit-test framework. I then provide some of my experience using JAVACOP as a discipline checker via a case study of a number of pluggable type system implementations.

Chapter 4 gives further details about the CLARITY semantic type qualifier framework for C. In this chapter I overview the CLARITY rule language for defining type qualifiers and explain how rules are checked on C programs. I then provide and analyze a formal type system and constraint inference algorithm for the type qualifiers. This is followed by an explanation of how rule specifications are verified using an automatic theorem prover. I conclude the chapter with a discussion of my experience

using a variety of CLARITY type qualifiers on real C code.

Chapter 5 concludes the paper with a look towards future extensions of this work.

CHAPTER 2

Related Work

In this chapter, I present and discuss related work to this dissertation. This work includes: formal specification and advanced type systems which allow programmers to precisely define their desired program properties but do not provide general strategies for how to establish such properties; language extensions that build support for a fixed set of disciplines into their type systems; and type system extension frameworks that allow programmers to define their own pluggable type systems but do not provide a discipline language for creating disciplines. I also briefly describe other work that shows how discipline checking and user-defined discipline frameworks fit into the context of broader research concerns.

2.1 Formal Specification

Formal specification is a common method of documenting the desired set of properties and behaviors of a program. Such specification is typically in the form of pre-condition, post-condition, and invariant specifications in the manner of Hoare Logic [Hoa69] or Design by Contract [Mey92]. A number of languages, such as Eiffel [ISO] and Spec# [BDF08], have built-in support for this kind of specification. Other languages, like Java, can utilize externally defined languages such as JML [LLL99] to allow programmers to write specifications in program comments. These specification comments need to be specially parsed and handled by other tools as they will not be

recognized by the target language.

While formal specification allows developers to precisely define desired program properties, it is not practical to check such specifications. Many formal specifications cannot be automatically checked against an actual program. However, languages like Spec# and tools such as ESC/Java [FLL02] (which utilizes JML specifications) provide support for automated reasoning over a limited domain of properties.

The formal specification approach is distinct from discipline checking in that specifications do not provide any guidelines to a programmer about how to write programs that match the formal specification. In this fashion, specifications are akin to programming goals, whereas programming disciplines describe programming strategies for ensuring program properties. In other words, disciplines define *how* to achieve certain program properties whereas formal specifications state *which* program properties are intended.

2.2 Advanced Type Systems

Some type systems, including the calculus of constructions [CH88], Nuprl [CAB86], and type systems [SST02, CV02] for Proof-Carrying Code [Nec97] and Typed Assembly Language [MWC99], use dependent types [Mar82] to encode program invariants directly in a program's types. However, dependent type checking is not as simple as checking disciplines, which are syntactic program constraints: type checking means that the invariants must be proved valid on their associated program expressions. A proof that an invariant holds on a program fragment cannot, in general, be produced automatically. Many of the proofs need to be supplied by the programmer. However, it is not reasonable to assume that the average programmer will have the knowledge to create such proofs.

Dependent type systems are very expressive, and since the programs contain their own proofs of correctness, are also inherently verifiable. However, as programming in a dependently-typed language is more complicated, usability is severely reduced. With the discipline approach, I provide usability by separating the simple and purely syntactic discipline checking from discipline verification.

Dependent types can be made more practical by limiting the grammar of the dependent types. For example, Dependent ML [XP98, XP99] allows ML types to depend upon integers with linear inequality constraints. This limited form of dependent types can be used to automatically prove arithmetic program invariants, such as tracking the length of lists or whether a variable contains a positive value.

2.3 Extensible Compiler Frameworks

Extensible compiler frameworks, such as JastAdd [EH04] and Polyglot [NCM03] for Java and CIL [NMR02] for C, can be used to extend a language with new functionality. This functionality can include new program analyses in the compiler and extensions to the syntax and semantics of the language. A developer could use such an extensible compiler framework to directly implement a discipline checker. However, these frameworks do not provide language-level support for writing disciplines. To create a discipline checker in such a framework, the developer must first be well-acquainted with compiler construction. The average programmer does not have this compiler expertise. Extensible compiler frameworks are useful for creating discipline frameworks, though. CLARITY, for instance, was developed on top of CIL.

2.4 Language Extensions

Language extensions that support a fixed set of additional disciplines are commonly built in the research community to allow researchers to apply new disciplines and type systems to benchmarks and real-world programs. Examples of language extensions include CCured [NMW02], which builds a pointer-safety discipline into C; Cyclone [GMJ02], which provides new syntax to support safe memory management in C; and RCCJava [FF00], which provides a locking scheme discipline to prevent race conditions in Java. I hope that discipline frameworks can provide the implementation platform for future development of new disciplines instead of creating a language extension from scratch. This would encourage other programmers to use new disciplines in their code as they would not have to migrate their entire code base to a new language.

2.5 Standalone Discipline Checkers

Standalone discipline checkers can be used to statically check a fixed set of disciplines on programs without requiring a language extension. Many of these tools have seen widespread use in industry. For example, the FindBugs system [HP04] includes a non-null type checker, and Ma and Foster provide a tool for detecting unique references [FM07]. These standalone checkers are not as adaptable as the discipline frameworks I propose in this dissertation. If a user wants to modify how a discipline is checked in a static analysis tool, he or she has to directly change the source code of the tool. In contrast, disciplines written in a discipline language can be more easily modified to suit a programmer's preferences. Further, standalone checkers usually require whole-program analysis in place of requiring programmers to annotate their code. This approach is, thus, not suitable for modularly enforcing disciplines.

2.6 Constraint Checking Frameworks

Constraint checking frameworks like XIRC [ESM05, EMO04], JQuery [JV03], SemmleCode [HVM06], CCEL [DMR92] and ASTLog [Cre97] provide programmers the ability to statically check their own constraints on a program. These frameworks provide a domain-specific constraint language that treats the target program as a database to be queried. Checking constraints in these frameworks requires whole program analysis and lacks interaction with the underlying type system. As a result, they are ill-suited for discipline checking. Further, none of these frameworks provides any support for verifying constraints against desired program properties.

The Structural Constraint Language (SCL) [HH06] for Java provides similar constraint checking capabilities to a subset of the JAVACOP language (described in Chapter 3). The language provided by SCL is similar to a straightforward first-order logic language, but distinguishes itself from the previous constraint checking frameworks by checking constraints modularly. However, due to limited integration with the Java type system and a lack of support for flow-sensitivity, it cannot check general purpose type systems like non-null or confined types.

The CoffeeStrainer constraint framework for Java [Bok99] eschews a constraint language in favor of having programmers write their constraints as Java methods inside special program comments. When run through the CoffeeStrainer tool, these methods will be extracted into a new visitor class. The visitor can then be used to check the constraints during compilation of the class. Constraints in CoffeeStrainer may only written at the class level, and cannot be written for fields or methods. While such an approach can handle class-level type systems like *confined* from Chapter 1, which was originally implemented using CoffeeStrainer [VB99], it would not be able to effectively handle the *untainted* discipline defined in the same chapter or any of the other examples presented in this dissertation.

2.7 Type System Extension Frameworks

Papi *et al.* created the Checker framework [PAC08] for developing pluggable type systems for Java. The goals of the Checker project are similar to the goals of this dissertation: they want to allow developers to write their own type systems and automatically type check them on their code. In the Checker framework, developers must use Java to write visitors that traverse Java's `Tree` classes, while this dissertation advocates providing a declarative rule language for such frameworks. The Checker framework provides no special support for writing advanced type systems that require dataflow analysis. It further lacks any facility for determining whether a pluggable type system ensures desired program properties. The Checker framework does, however, support inference of certain forms of information flow qualifiers, along the same lines as the simple form of the *untainted* discipline that was described in Chapter 1. Initial papers on both the CLARITY and JAVACOP projects presented in this dissertation predate the publication of the Checker framework.

The Vault language provides a type system that tracks temporal protocols in a safe dialect of C [DF01]. A type in Vault can be augmented with a *type guard* that tracks the state of the value throughout the program. For example, a file type could be augmented with a guard that tracks whether the file is currently in an open or closed state at any given point in a program. The states associated with a guard as well as state-change conditions are defined by the programmer. Vault's type system is well-suited for checking simple temporal disciplines such as open/closed file states and device driver states, but it is not expressive enough to handle other general-purpose or domain-specific disciplines.

Fugue [DF04] is an adaptation and extension of Vault's type system to perform static checking of temporal protocols for C# [C02]. Fugue allows a class's typestates, which are the analogues of type guards, to be given an interpretation as a predicate

over the class's fields. Such predicates are used during static type checking to ensure that each method in the class properly implements its declared specification. In this way, the Fugue type checker directly ensures that typestates respect their predicates. This is the same kind of verification of type checking against invariants that I propose for user-defined discipline frameworks.

The CQUAL framework [FFA99, FTA02] allows programmers to define and enforce partial orders of *type qualifiers* on C programs. Type qualifiers in this case are lightweight type annotations. Such type qualifiers can be used to check simple forms of disciplines such as the untainted discipline from Chapter 1. However, CQUAL does not support general syntactic constraints; a limited form of syntactic constraints may be simulated in CQUAL via qualifier assertions and assumptions. CQUAL also does not support any kind of reasoning about the validity of a qualifier with respect to its intended invariant. More recently, the authors of the CQUAL project have extended this notion of simple type qualifiers to object-oriented programming via the JQUAL system [GF07]. CQUAL supports polymorphic, flow-sensitive qualifier inference. The flow-sensitive version of CQUAL can be used to track similar kinds of temporal properties as Vault without requiring the same amount of program annotation.

2.8 Program Property Inferencers

Program property or invariant inferencers can be thought of as the dual to discipline checking frameworks. Whereas discipline checking ensures certain desired program properties by enforcing syntactic constraints, invariant inference can find program invariants from otherwise undisciplined code. Such inference can occur either statically or dynamically.

The Daikon [ECG01] and Diduce [HL02] systems both support dynamic program

invariant inference. Both of these systems take a source program and instrument it to gather data at runtime. Daikon infers invariants by running the instrumented program on a large test suite and using the gathered data to determine the strongest invariants for integer and sequence variables. Diduce refines invariants on the fly and keeps track of when inferred invariants are violated; these instances are reported as potential errors in the program. The invariants that result from these inferencers can be useful for adding or refining formal specifications to programs. Likewise, they might inspire a programmer to use a discipline which ensures the invariants remain valid with future program updates.

2.9 Provably Correct Compiler Optimizations

One of the primary inspirations for this work comes from the Cobalt and Rhodium systems [LMC03, LMR05]. These frameworks provide a domain-specific language which allows compiler writers to write dataflow analyses and optimizations. These compiler analyses and optimizations can be automatically proved sound by discharging proof obligations with an automatic theorem prover. This overarching model was adapted in this dissertation to allow general programmers to easily extend their type systems to enforce new programming disciplines. In particular, the idea to separate discipline specification and discipline validation comes from these earlier systems.

CHAPTER 3

JAVACOP: Pluggable Types for Java

3.1 Introduction

In this chapter, I present the design, implementation, and evaluation of a practical framework for pluggable type systems in Java, which I call JAVACOP. This framework was built to demonstrate a user-defined programming discipline checking framework that offers a high level of discipline expressiveness. To this end, I have designed a declarative rule language in which programmers specify their pluggable type systems. User-defined rules function over a rich abstract syntax tree (AST) representation for Java programs and can use Java metadata annotations [Blo02] to introduce new type information into the source code. JAVACOP enforces these user-defined rules on Java programs as they are compiled.

As a simple example, consider the code in Listing 3.1 which uses an `@Untainted` annotation to specify the additional type constraint that the field `firstname` (of Java type `String`) is safe to use when constructing SQL queries. The assignment to `firstname` in the method `setFirstName` potentially violates this constraint, since the parameter `newname` could be a tainted `String`. In the case that `"Shane';DROP TABLE users;--"` is passed as the argument, using `firstname` to construct a SQL query could result in a difficult-to-trace SQL injection attack.

It is straightforward for a user to write a JAVACOP rule that would discover this potential error. For example, the JAVACOP rule in Listing 3.2 requires the right-hand-

```
class Person {
   @Untainted String firstname = "Chris";
   void setFirstName(String newname) {
      firstname = newname;
   }
}
```

Listing 3.1: A Java class demonstrating a potential vector for a SQL injection attack.

```
rule checkUntainted(Assign a){
  where(requiresUntainted(a.lhs)){
    require(definitelyUntainted(a.rhs)):
      error(a,"Possible tainted assignment"
            + " to @Untainted");
}}
```

Listing 3.2: A JAVACOP rule which prevents assignment of a possibly tainted value into an `@Untainted` reference.

side expression of each assignment statement to be demonstrably untainted whenever the type of the left-hand-side variable or field is declared as such.

This rule relies on two user-defined helper predicates. The `requiresUntainted` predicate checks that the given variable or field was declared with the `@Untainted` attribute. The `definitelyUntainted` predicate inspects the given expression to determine if it is definitely untainted. For example, `definitelyUntainted` would define `String` literals to be definitely untainted as the literal is always under the control of the developer. Assuming `definitelyUntainted` does not allow an arbitrary variable to be considered untainted, the assignment to `firstname` in `setFirstName` will fail the rule, causing the compiler to output the error message below.

```
Person.java:4: Possible tainted assignment to @Untainted
    firstname = newname;
    ^
1 error
```

JAVACOP provides a novel combination of features to support the development of practical pluggable type systems:

- *Declarative rule language.* JAVACOP employs a declarative, rule-based language for expressing the semantics of pluggable type systems. JAVACOP's language was created to be a simple yet highly expressive language for defining new Java type system extensions that is easy for rule designers and programmers to understand and to define correctly. The language has a natural correspondence to the normal specification of syntax-directed typing rules and to first-order logic programming.

- *Seamless integration with Java.* JAVACOP naturally allows pluggable type systems to interact with Java's existing type system, including information about generics and annotations. It retains Java's modular style of typechecking and compilation. The typechecker is implemented as an extension of the standard OpenJDK `javac` compiler with a couple of new passes. This feature allows the JAVACOP compiler to be used in place of the standard Java compiler in a development toolchain.

- *Support for flow sensitivity.* Many pluggable type systems require the ability to change the type of a variable based on its context. For example, a non-null type system should allow a possibly-null variable `x` to be considered non-null after a successful test of the form `x != null`; if not, the type system will be too inflexible for practical usage. To provide support for these pluggable type systems, JAVACOP includes a generic dataflow engine that can be extended by the user to generate necessary dataflow facts. These dataflow facts can then be referenced in JAVACOP rules as an addition to the type system.

- *Pluggable type system validation.* To address discipline verification, JAVACOP

provides a novel two-stage framework for testing pluggable type systems against a test suite of Java programs. The first stage runs the JAVACOP checker on each test program and compares against the expected typechecking outcomes, similar to traditional unit testing. However, many pluggable type systems are meant to enforce a simple set of runtime invariants. For example, a non-null type system should ensure that a `@NonNull` variable or field never has the value `null`. The second stage of JAVACOP's testing framework executes instrumented versions of successfully-compiled test programs to verify that the intended runtime invariants are not violated. Such testing is not complete, but it does provide some idea of the soundness of the discipline for the given test suite.

I have built a diverse suite of complete and practical pluggable type systems in JAVACOP and used them to detect real errors in existing Java software. Based on JAVACOP compilation performance tests, I believe that use of JAVACOP checkers causes minimal overhead, allowing it to be used during interactive development. JAVACOP and all associated tools were released under the open-source GNU General Public License v2.0.

3.1.1 Overview of the JAVACOP Framework

Fig. 3.1 provides an overview of the structure of the JAVACOP system. The JAVACOP type checker has two main components: a Java compiler extended with JAVACOP pluggable type support and a JAVACOP language parser/compiler. In addition to the main system, there are a few additional tools that make up the JAVACOP tool suite: a dataflow analysis API, a Java runtime instrumenter (not shown in figure), and a pluggable type system unit test framework.

The JAVACOP engine is an extension of the OpenJDK Java compiler which takes as input compiled JAVACOP rules and dataflow analyses as well as target programs.

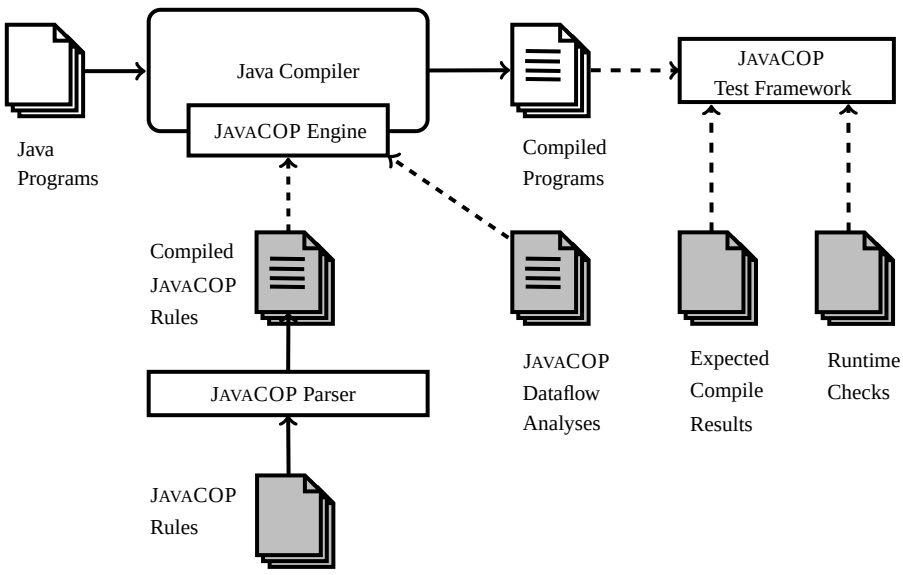

Figure 3.1: An overview of the JAVACOP system. Dashed lines indicate optional inclusions; and document images with content represent Java bytecode.

JAVACOP then runs the dataflow analyses and does type enforcement on the target Java programs. These passes occur after the Java AST has been attributed with type information but before any substantial optimizations or compilation has occurred. If there are no errors signaled during either type attribution or the JAVACOP passes, compilation continues as expected.

Dashed lines in Figure 3.1 indicate optional argument inclusions, including any JAVACOP pluggable types and dataflow analyses. Compiled Java code is shown as an image of a document with text. One design choice for the JAVACOP framework was to separate the parsing and compilation of the typing rules from enforcing the rules via the JAVACOP Java compiler. This decision cuts down on the time required to do the type system enforcement. Use of the JAVACOP test framework is always optional.

The pluggable type system engine is an extension of the OpenJDK `javac` compiler, version 1.7.0-ea. As a result, interfacing with the JAVACOP engine is simply an

additional option to the `javac` command which takes a set of compiled JAVACOP files as its argument. The extended compiler still takes standard Java programs as input, but prints the warnings and errors generated by the pluggable type systems in addition to the standard `javac` output.

3.1.2 Chapter Organization

The rest of this chapter is organized as follows. Section 3.2 introduces the design of JAVACOP's rule language through a number of examples. Section 3.3 provides a formal semantics for JAVACOP through a translation of the core language to the Datalog$^\neg$ language. Section 3.4 demonstrates JAVACOP's facilities for flow-sensitive reasoning. Section 3.5 describes the pluggable type system testing framework. Section 3.6 shows how JAVACOP can be used to develop two domain-specific pluggable type systems developed in JAVACOP, showcasing the expressiveness of JAVACOP's declarative rule language. Section 3.7 presents experience building and using a more sophisticated general-purpose pluggable type system that leverages flow-sensitivity. Section 3.8 presents compiler performance test numbers.

3.2 The JAVACOP Rule Language

This section describes JAVACOP's rule language in detail. The first part describes some of the features of the AST representation of programs that the language employs. The rest of the section describes the features of the JAVACOP rule language, and their utility in implementing pluggable type systems, via several examples.

Tree subclass	Name	Java example
`JCMethodInvocation`	Method call	`meth(args)`
`JCAssign`	Assignment	`x=y`
`JCClassDecl`	Class definition	`class X{ .. }`
`JCIdent`	Identifier	`foo`
`JCIf`	Conditional	`if(cond).. [else ..]`
`JCMethodDecl`	Method definition	`void foo(){..}`
`JCNewClass`	Instance creation	`new World("hello")`
`JCReturn`	Return statement	`return false;`
`JCFieldAccess`	Field Selection	`s.field`
`JCSkip`	Empty statement	`;`
`JCTypeCast`	Cast	`(String)s`
`JCVariableDecl`	Variable declaration	`String s = "hello";`

Figure 3.2: A selection of OpenJDK AST nodes classes and their meanings

3.2.1 The Abstract Syntax Tree

The AST of a Java program is made up of linked nodes representing the program's structure: classes, methods, blocks, statements, expressions, identifiers, etc. JAVACOP's AST is an abstraction of the OpenJDK compiler AST, in which all the node types are subclasses of the abstract superclass `JCTree`. Figure 3.2 lists a selection of these AST nodes and the Java code they represent. Each node provides methods and fields to access its sub-nodes.

JAVACOP's traversal of the AST occurs as a pass after Java typechecking has occurred. This allows JAVACOP rules to make use of Java type information: this is critical for many kinds of type extensions. Every node in the AST contains a `type` field of type `Type`, which is set during the typechecking pass and represents the Java type of

the expression represented by the node. These types include class types (which may be parametrized), array types, method types, and (bounded) type parameters; Java interfaces are represented by class types internally. Types have methods that allow their component types to be accessed, and class types have methods that allow their direct superclass (`supertype()`) and superinterfaces (`interfaces()`) to be retrieved.

In order for a class to be compiled, `javac` requires information about each non-local identifier — package, class, interface, method, and field name — that is referenced in the class. If `javac` were a whole-program compiler, each identifier could simply be linked to the AST node for its associated definition. Given `javac`'s modular compilation, the source of some depended-upon program entities and the AST nodes for those entities, may be unavailable. The `javac` compiler reconstructs necessary information about non-local entities from their bytecode representations, and stores it as `Symbol` objects.

The JAVACOP AST is currently built on top of the AST representation in `javac`, but there is a separation between the two. For example, JAVACOP AST nodes include several useful methods that are not directly available in the underlying `javac` nodes. These methods provide a JAVACOP developer with information that might be difficult or impossible to locate given only a `javac` AST node or require the use of reflection. Because of this extra layer of abstraction, it is possible to move the JAVACOP system to other Java compilers without changing the core rule language.

3.2.2 Rule Language Overview

Figure 3.3 presents the syntax of the JAVACOP rule language. A pluggable type system is implemented in JAVACOP as a set of *rules*, which constrain classes via the AST representation described in the previous subsection. Each rule is declared to apply to a particular kind of AST node and provides constraints on the usage of that node, de-

pending on type information, user-defined annotations, and other context information available at that node. To enforce user-defined rules, JAVACOP performs a depth-first traversal of the AST of a given compilation unit (i.e., a Java file). As each node is traversed, any rules that apply to that type of node are evaluated on that node.

As shown in Figure 3.3, there are two kinds of "joinpoint" for JAVACOP rules which determine when a rule is applicable: *AST* joinpoints and *subtyping* joinpoints. These joinpoints are detailed in the next two subsections. I will then discuss the various kinds of constraints that may be employed within a rule body, the `declare` construct for defining helper predicates for use within a rule, and JAVACOP's facilities for error reporting.

3.2.3 AST Rules

The first kind of JAVACOP rule in Figure 3.3 is a function that starts with the keyword `rule` and includes a name, a single parameter whose type is a (subtype of) `Tree`, and a body containing a sequence of constraints. When JAVACOP's AST traversal visits a node, the node is passed to each rule that takes an argument of the node's type. For example, the `checkUntainted` rule defined in Listing 3.2 will be passed each node representing a Java assignment statement during JAVACOP's traversal of an AST. As a result, each assignment statement is forced to satisfy the constraints in the body of `checkUntainted`. There are several forms of constraints, which are discussed in detail in later subsections.

Another simple example of this kind of rule is shown in Listing 3.3. This rule ensures that a `@NonNull` annotation is only used on variables and fields which have a reference type. While not necessary, using a rule like this prevents abuse of the `@NonNull` annotation on fields and variables that cannot ever be null by definition. The rule relies on the ability to look up the type of an AST node directly from the node

⟨*RuleFile*⟩ ::= (⟨*Rule*⟩ | ⟨*Declaration*⟩)+

⟨*Rule*⟩ ::= ‘rule’ ⟨*Identifier*⟩ ‘(’ ⟨*Joinpoint*⟩ ‘)’ ⟨*StatementList*⟩ ⟨*FailureClause*⟩?

⟨*Declaration*⟩ ::= ‘declare’ ⟨*Identifier*⟩ ‘(’ ⟨*VarDefList*⟩ ‘)’ ⟨*StatementList*⟩

⟨*StatementList*⟩ ::= ‘{’ ⟨*Statement*⟩+ ‘}’

⟨*JoinPoint*⟩ ::= ⟨*VarDef*⟩

| ⟨*Identifier*⟩ ‘<<:’ ⟨*Identifier*⟩

⟨*VarDefList*⟩ ::= ⟨*VarDef*⟩ (‘,’ ⟨*VarDef*⟩)*

⟨*VarDef*⟩ ::= ⟨*TypeIdentifier*⟩ ⟨*Identifier*⟩

⟨*FailureClause*⟩ ::= ‘:’ (‘error’ | ‘warning’) ‘(’ ⟨*Expression*⟩ ‘,’ ⟨*Expression*⟩ ‘)’

⟨*Statement*⟩ ::= ⟨*Condition*⟩

| ⟨*Quantification*⟩

⟨*Condition*⟩ ::= (‘where’ | ‘require’) ‘(’ (⟨*VarDefList*⟩ ‘;’)? ⟨*Expression*⟩ ‘)’

⟨*ConditionRest*⟩

⟨*ConditionRest*⟩ ::= ⟨*FailureClause*⟩? ‘;’

| ⟨*StatementList*⟩ ⟨*FailureClause*⟩?

⟨*Quantification*⟩ ::= (‘forall’ | ‘exists’) ‘(’ ⟨*VarDef*⟩ ‘:’ ⟨*Expression*⟩ ‘)’

⟨*StatementList*⟩ ⟨*FailureClause*⟩?

Figure 3.3: A subset of the JAVACOP Syntax. Expression syntax is not presented here, but handles most Java expressions and additionally supports let binding and pattern matching (Section 3.2.7.

```
rule checkNonNullRef(JCVariableDecl v){
  where(requiresNonNull(v)){
    require(!v.type.isPrimitive()):
      error(v,"@NonNull can only annotate variables of "+
              "reference type");
  }
}
```

Listing 3.3: A JAVACOP rule that ensures the `@NonNull` annotation is only used on subtypes of `Object`.

itself.

In a traditional type system, a program is considered to typecheck successfully if there is some way to derive a type for the program through the given typechecking rules. Because pluggable type systems in JAVACOP often impose only a few additional constraints onto the existing Java type system, JAVACOP enforces the opposite convention. In particular, a program (or compilation unit) successfully typechecks by default in JAVACOP, and JAVACOP rules are used to impose additional requirements to be satisfied. However, it is easy for a JAVACOP user to implement the traditional style if desired; an example is shown in Section 3.2.6.

3.2.4 Subtype Rules

A hallmark of most object-oriented type systems is the notion of subtyping. Static typechecking ensures that this subtyping relation is respected, such that values of a given type can only ever be *viewed* as that type or a supertype. For example, the type of the right-hand side in an assignment statement must be a subtype of the type of the reference being assigned into, and the actual arguments to a method call must be subtypes of the corresponding formal argument types. Pluggable type systems for object-oriented languages may need to extend the existing subtyping relation, in order

to prescribe the ways in which the new user-defined type specifications interact with other types–both Java types and user-defined ones.

JAVACOP's rule language supports the declarative specification of user-defined subtyping relationships. For example, the rule in Listing 3.4 subsumes the `checkUntainted` rule in Section 3.1 by ensuring a potentially tainted expression can never be treated as `@Untainted`. The only syntactic difference between writing a rule for subtyping and writing an AST rule, as described in Section 3.2.3, is the parameter list of the form `a <<: b`. The rule applies to any AST node where a subtype relationship is traditionally required, including assignment statements, return statements, parameter passing, and type casts. JAVACOP subtyping rules receive the expression `a` whose type must be compatible with the type of the symbol `b`. At an assignment node, for example, the right-hand-side expression will be checked against the symbol for the variable or field being assigned into.

Subtyping rules can also be written that define supertypes of an unqualified Java type. For example, the goal of a type system for confinement [VB99] is to ensure that instances of types that are marked as *confined* are only accessible within the type's defining package. To achieve this goal, such a type system imposes a number of restrictions on confined types. The rule in Listing 3.5 enforces one such restriction, which prevents a value of confined type from being treated as an unconfined supertype. The `confined` helper predicate used in the rule checks whether a type was declared `@Confined`.

3.2.5 Constraints

The body of a JAVACOP rule consists of a sequence of constraints. The basic kind of constraint has the form "**`require`**`(<condition>);`". Such a constraint is satisfied if the associated boolean condition evaluates to `true`; otherwise the constraint fails. As a

```
rule checkUntainted(node <<: sym){
  where(requiresUntainted(sym)){
    require(definitelyUntainted(node)):
       error(node, "Possibly tainted expression "+node
                   +" used where @Untainted expected");
}}
```

Listing 3.4: A rule which enforces proper subtyping constraints for an *untainted* type system.

```
rule checkConfined(a <<: b){
  where(confined(a.type)){
    require(confined(b.type)):
      error(a, "Confined expression may not be treated "
               +"as a subtype of unconfined type "
               +b.type);
}}
```

Listing 3.5: A rule for a *confined* type system which does not allow a confined type to be converted into an unconfined type.

simple example, the rule in Listing 3.6 shows how JAVACOP can encode the semantics of the `final` modifier for Java classes.

This rule checks each class definition to ensure that the class does not inherit from a class that has the `@Final` attribute. The constraint in the rule employs the rule language's `Symbol` objects to access *interface* information about a class's declared superclass. The JAVACOP rule language ensures that rules can be enforced modularly via the API that it provides to users. For example, while it is possible to access a given class's superclass, it is not possible to access all subclasses of that class. This modular style of checking can be seen in the construction of the `finalClass` rule as the global property that a `@Final` class must have no subclasses is ensured by individual checks on each compiled class of a project.

Often a constraint should only be applied under certain circumstances. This can be

```
rule finalClass(JCClassDecl c){
  require(!c.supertype.sym.hasAnnotation("Final")):
    error(c, "Superclass has a @Final annotation!");
}
```

Listing 3.6: A JAVACOP rule that enforces the `final` modifier on Java classes by signaling an error when a subclass extends a final class.

accomplished through the use of a `where` constraint. Like `require`, a `where` constraint has a boolean guard condition. In addition, a `where` constraint has a body containing a sequence of other constraints. The `where` constraint is satisfied if either its guard evaluates to `false` or all constraints in the body evaluate to `true`. An example `where` constraint is shown in the `checkUntainted` rule from Listing 3.4. In that rule, the right-hand-side expression of an assignment need only be shown to be untainted if the left-hand-side variable or field is annotated with `@Untainted`.

The language for constraint conditions is an extension of the language for Java `boolean`-typed expressions. These expressions can invoke methods on any AST nodes, types, and symbols in the scope of the constraint. JAVACOP also supports conditions that perform let-binding type tests and structural pattern matching, which are discussed in Subsection 3.2.7.

JAVACOP includes support for Java's primitive types and `String`, and their associated operations and methods, as well as for a `List` type provided by `javac`. Constraints may also employ values of two new types: a *traversal environment* `Env` holds information about the tree context surrounding a given node, and a *global environment* `Globals` is a repository for global constants, such as the type objects for `java.lang.Object` and primitive types, and the symbols for the root and empty packages. An instance of each of these two types is implicitly in scope in each rule, with the name `env` and `globals` respectively. Some examples in the next section illustrate their usage.

Finally, the developer may directly look up fields or invoke methods on Java objects which are not natively supported by the JAVACOP API by using the # operator. While I believe that the JAVACOP language and API suffice for most type systems, the ability to use an API defined outside of the one provided by JAVACOP is occasionally useful. For instance, the # operator is used to allow JAVACOP rules to call into a user-defined dataflow API built on top of the JAVACOP flow-sensitive framework, discussed in Section 3.4.

3.2.6 Auxiliary Predicates

In addition to rules, JAVACOP allows users to declare auxiliary predicates, analogous to the auxiliary predicates sometimes used in formal type systems (e.g., Featherweight Java's override and downcast [IPW01]) and first-order logic, using the `declare` keyword. These predicates are not tested directly during JAVACOP's AST traversal but instead are used simply as helpers for rule definitions. Predicates are invoked by the bodies of rules and other predicates using a traditional function-call syntax. For example, the rule `checkUntainted` from Section 3.2.4 makes use of a `requiresUntainted` predicate which is defined in Listing 3.7.

The `requireUntainted` predicate gets the given node's associated symbol, if it has one, and uses the symbol to check whether the node has the appropriate annotation. Auxiliary predicates provide the usual benefits of procedural abstraction. In this case, the `requiresUntainted` predicate serves to separate the logic that determines how *untainted-ness* is annotated from the logic that determines the behavior of programs employing such an annotation. This separation makes it easy to augment or modify the annotation mechanism. For example, untainted-ness could be indicated by using a naming scheme instead of Java's metadata facility, simply by changing the definition of the `requiresUntainted` predicate as in Figure 3.8.

```
declare requiresUntainted(JCTree t){
  require(t.holdsSymbol
          && t.getSymbol.hasAnnotation("Untainted"));
}
```

Listing 3.7: A predicate which returns true when the checked tree has been marked with the `@Untainted` annotation at declaration.

```
declare requiresUntainted(JCTree t){
  require(t.getName.startsWith("untainted_"));
}
```

Listing 3.8: A predicate which returns true when the checked tree has been given a name starting with the `String "untainted_"`.

Rule and predicate bodies naturally support a form of conjunction for constraints, by sequencing multiple constraints. Predicates themselves additionally provide a declarative form of disjunction. Following the conventions of type system definitions, JAVACOP allows an auxiliary predicate to have multiple definitions; an invocation of the predicate succeeds if at least one of the definitions' bodies is satisfied.

For example, the `checkUntainted` rule from Listing 3.4 makes use of the `definitelyUntainted` predicate, which checks whether a given `JCTree` object is definitely untainted. This predicate can be defined in JAVACOP as a case analysis on different subtypes of `JCTree` by providing multiple definitions of the predicate. A few representative definitions are provided in Listing 3.9. A `declare` implicitly performs a type test on a given node against the declared type of its argument. This type test desugars into a `require` constraint: the `declare` definition fails if the type test fails. For example, the third definition in Listing 3.9 fails if the given node does not represent a Java assignment expression.

As mentioned in Section 3.2.3, predicates also allow developers to define type systems in a more traditional fashion where every type-correct expression must be

```
/* A node is untainted if annotated as such */
declare definitelyUntainted(JCTree t){
  require(requiresUntainted(t));
}

/* A literal is always untainted */
declare definitelyUntainted(JCLiteral l){
  require(true);
}

/* The value of an assignment is untainted
   if the value being assigned is untainted */
declare definitelyUntainted(JCAssign a){
  require(definitelyUntainted(a.rhs));
}
```

Listing 3.9: Three definitions of a predicate which indicates whether an expression should be considered untainted in all cases.

defined explicitly, as opposed to the allowed-by-default semantics of JAVACOP rules. For instance, the developer could define a subtyping predicate for untainted that must hold at all places where subtyping is required. Instead of defining the subtyping rule as in Listing 3.4, the rule could instead be defined primarily via a predicate as shown in Listing 3.10. Unless one of the `isSubtype` predicate definitions is satisfied, this rule will fail.

3.2.7 Pattern Matching and Conditional Assignment

Type rules often depend on the ability to deconstruct the expressions, types, and environments that they constrain, so it is imperative that a language designed for encoding pluggable types also have this ability. To that end, JAVACOP supplies two new operators: `<-` for type-conditional value binding, and `=>` for pattern matching on AST nodes.

```
rule checkUntainted(node <<: sym){
  require(isSubtype(node, sym))
    :error(node, "Subtyping constraints violated for "
                 +"untainted type system");
}

declare isSubtype(JCTree node, Symbol sym){
  require(!requiresUntainted(sym));
}

declare isSubtype(JCTree node, Symbol sym){
  require(requiresUntainted(sym)
          && definitelyUntainted(node));
}
```

Listing 3.10: A replacement rule for `checkUntainted` from Listing 3.4 which requires that *all* subtyping nodes satisfy the provided `isSubtype` predicate.

An expression of the form `v <- e` evaluates to true if `e` is found to be an instance of the type of the variable `v`. If satisfied, `e` is cast to the type of `v`, which is then let-bound to this value. Otherwise, the type-conditional binding evaluates to false and the value is not bound. A constraint condition may be preceded by a list of variable declarations to be bound within its conditional expression. For example, the following constraint binds the reference and field/method begin accessed at a explicit dereference point (`dref`) to the `recv` and `mesg` variables:

```
require(Tree recv, String mesg;
        recv <- dref.selected && mesg <- dref.name){...}
```

As another example, in many type systems, such as the non-null type system described in Section 3.4, it is necessary to distinguish between the explicit `this` receiver and other receivers at method invocation sites. The predicate `explicitThis` in Listing 3.11 uses type-conditional assignment conditions in order to test this property. Given a method invocation AST node, it uses conditional binding to first match only receivers that do an explicit dereference and then to match only trees with a single ex-

```
declare explicitThisReceiver(JCMethodInvocation m) {
  require(JCFieldAccess s ; s <- m.meth) {
    require(JCIdent i ; i <- s.selected) {
      require(i.name.equals("this"));
}}}
```

Listing 3.11: A JAVACOP predicate that is satisfied when checked on a method call with an explicit `this` as its receiver. This predicate is used in the non-null checker further discussed in Section 3.4.

plicit dereference. If both conditional bindings succeed, then the explicit dereference is required to be equal to the `this` identifier.

JAVACOP also includes an expression sub-language for pattern matching on AST nodes. Pattern matching allows for declarative testing of properties of an AST node, while also deconstructing the node and giving names to its component nodes for use in the rest of a constraint. A pattern match is a `boolean` expression: `e => [ pat ]`. In this expression, `e` is an arbitrary expression of type `Tree`, and the pattern match succeeds if the value of `e` can successfully be matched against the pattern `pat`.

Patterns are written as fragments of Java code which must be structurally equivalent to a targeted expression in order for the match to succeed. Patterns may include wildcard elements, which are written `^^` to match any subtree, `^*` to match any identifier name, and `...` for any number of elements in a sequence, such as statements in a block or parameters to a method. For example, the pattern:

```
v => [@Untainted ^^ ^*(...)]
```

matches a `JCTree` node `v` against a method declaration which has an `@Untainted` annotation, has any return type, any name, and any number of arguments.

Patterns may also bind `Tree` and `String` sub-components encountered in the pattern structure to fresh variables declared in the constraint. Variable binding involves an implicit type test: for a variable binding to succeed, the type of the component value

must meet the declared type of the new variable. If one of these variables is used more than once within the same expression, it is bound to the first instance, and the second and subsequent instances are compared according to reference equality. For example, variables can be bound to the subexpressions in the pattern described above as follows:

```
where(Tree typ, String name;
        v => [@Untainted typ name(...)])
```

Patterns may also test whether a given subtree is equivalent to the result of evaluating the JAVACOP expression between ‘ markers. For example, if `n` is a variable of type `JCTree`, then the following constraint requires `n` to be the first statement in the current location's enclosing block, which can be looked up from the context environment (`env.next`). If the first statement in the enclosing block does not equal `n`, then this pattern match will fail:

```
require(env.next => [{‘n‘;...}])
```

Pattern matching can significantly improve the readability and shorten the length of rules that would otherwise require multiple type tests. For example, the rule from Listing 3.11 can be rewritten using pattern matching as in Listing 3.12.

3.2.8 Quantification

JAVACOP provides quantification over two kinds of data structures. First, constraints may universally or existentially quantify over `javac` `List`s with `forall` and `exists` quantifiers, respectively. The syntax is similar to the syntax of the enhanced `for` in Java 1.5. For example, the predicate defined in Listing 3.13 requires that *confined* classes do not have *unconfined* supertypes through quantification over all supertypes. The `forall` iterates over a list of all types found on the list returned by `c.supertypes()`, binding each to the name `s` in turn. The variable used to bind each element of a list must have the same type as the declared element type of the list. The syntax for existential

```
declare shortExplicitThisReceiver(JCMethodInvocation m){
  require( m.meth => [this.^*] );
}
```

Listing 3.12: A more succinct version of the JAVACOP rule from Listing 3.11, which uses pattern matching.

```
rule ConfinedDef(JCClassDecl c){
  where(!confined(c)){
    forall(Type s : c.supertypes()){
      require(!confined(s)):
        error(c,...);
}}}
```

Listing 3.13: A JAVACOP rule–demonstrating quantification over lists–which requires that *confined* classes do not extend *unconfined* superclasses or implement *unconfined* interfaces.

quantification is identical, except that it uses the keyword `exists` instead of `forall`.

Second, JAVACOP allows quantification over all nodes in a depth-first traversal from a given AST node. During this traversal, only nodes that match the declared type of the quantified variable are considered. For example, the predicate defined in Listing 3.14 determines if a field is used as the lock for another variable or field in the same class The predicate uses existential quantification to iterate over every variable and field declaration (`JCVariableDecl`) in the body of the class currently being visited (`env.enclClass`). The predicate also shows interaction between the advanced features of JAVACOP (quantification and pattern matching) to create concise and readable definitions.

```
declare isSynchLock(JCVariableDecl v){
  exists(JCVariableDecl v2 : env.enclClass){
    /* don't do pattern match if no lock declared */
    require(declaresLock(v2)){
      require(Literal n; v2 => [@LockedBy(n) ^^ ^*]){
        require(n.value.equals(v.name));
}}}}
```

Listing 3.14: A JAVACOP predicate–demonstrating quantification over AST nodes–which determines if a field is defined as a lock for another field or variable for use in a race condition checking type system.

3.3 JAVACOP Language Semantics

In this section, I present a semantics for a core subset of the JAVACOP language which I call FEATHERWEIGHTJAVACOP, or FJCOP, via a translation to the Datalog¬ language[CGT90]. Datalog¬ is one of the more widely used formal declarative query languages for relational databases, and is, thus, a good reference point for the expressiveness of the language. The purpose of this presentation is to define more formally *what* a JAVACOP program is and what kind of information about Java programs it can provide. I hope that this formalism may lead to future insight into how to make JAVACOP more efficient or how to automatically reason about JAVACOP programs via proof assistants or automated theorem provers.

3.3.1 Datalog¬

To understand the translation from FJCOP to Datalog¬, I will first provide a small overview of Datalog¬ that provides insight into its syntax and semantics. I use the definition of Datalog¬ as defined by Ceri et al. [CGT90] as the basis for the translation.

In Datalog¬, there are three infinite alphabets that make up the basic elements of Datalog¬ programs: variables, constants, and predicates. Predicates are simply an

n-ary mapping from literals to truth values.

A *term* in Datalog¬ is either a variable or a constant, an *atom* is an n-ary predicate and a list of arguments which are all terms, and a *literal* is either an atom or a negated atom. A *clause* is a set of literals. A Datalog¬ program is made up of a set of clauses of which there are three kinds: *facts*, *rules*, and *goals*. Facts consist of a single positive literal, rules consist of a head literal and a body of literals and are written as in Prolog $head :- body$, and goals contain only a body of literals written $?- body$.

There are two kinds of predicates in a Datalog¬ program: *extensional* predicates and *intensional* predicates. Extensional predicates are defined by the extensional database over which the Datalog¬ program will run; these predicates are defined before execution of the program. Intensional predicates are defined by the program itself; the execution of the program will generate facts regarding intensional predicates. For FJCOP, the extensional predicates will consist of predicates related to the Java AST and base type system whereas the intensional predicates will consist of the rules and predicates defined in the FJCOP program.

As Datalog¬ is a query language, the semantics of Datalog¬ are defined by a mapping from the powerset of extensional predicate ground clauses to the powerset of all predicate ground facts. It is easy to see that ground facts correspond to the values of this model. As facts strictly define new information, they provide their own semantics; a ground fact is already a value, and any fact with a variable is simply expanded into a powerset from the literals defined in the extensional database. A rule is a fact generator in that it can be used to derive new ground facts for an intensional predicate; thus, the semantics of a rule is a mapping from the extensional database to the set of ground facts for the predicate that the rule implicates. Adding a query to the program changes the output of the program to the sets of extensional and intensional ground facts which are subsumed by the query.

```
checkUntainted(A) :- type(A,JCAssign),F1(A).
F1(A) :- lhs(A,L),type(L,JCTree),F2(A,L).
F1(A) :- object(A),lhs(A,L),¬(type(L,JCTree)).
F2(A,L) :- requiresUntainted(L),F3(A,L).
F2(A,L) :- object(A),object(L),¬(requiresUntainted(L)).
F3(A,L) :- rhs(A,R),type(R,JCTree),definitelyUntainted(R).
F3(A,L) :- object(A),object(L),rhs(A,R),¬(type(R,JCTree)).
```

Listing 3.15: Datalog¬ Translation of JAVACOP program from Listing 3.2.

For Datalog¬ programs with negation in the body of rules and goals, this mapping is only well-defined for predicates with stratified negation (i.e., predicates that may be partitioned into strata that do not allow recursive dependence on negation). Ceri et al. [CGT90] provide more details on the Datalog¬ semantics as well as query execution strategies.

3.3.2 Translation Overview

The FJCOP language allows users to define rules and predicates over Java AST classes in a similar manner to a standard JAVACOP program. The translation scheme presented here shows this form of checking reduces to a satisfiability problem in Datalog¬. The FJCOP framework can be thought of as an extensional database (EDB) generator for a given Java AST. The FJCOP API provides the extensional predicates (EPred) which the translated FJCOP rules and predicates will use. The translation itself creates the intensional predicates (IPred) which are a super set of the predicates and rules explicitly defined in the FJCOP program. For example, Listing 3.15 shows a translation of the basic subtyping rule for assignment to an untainted reference as defined in Listing 3.2.

It also outputs a set of Datalog goals of the form

```
? :- type(X,τ),¬(f(X)).
```

$$
\begin{array}{rcl}
d & ::= & \textbf{declare}\ f(x:\tau+)\{c\} \\
 & | & \textbf{rule}\ f(x:\tau)\{c\} \\
 & & \\
t & ::= & x \mid l \\
 & & \\
c & ::= & c;c \\
 & | & \textbf{require}(e_b); \\
 & | & \textbf{where}(e_b)\{c\} \\
 & | & \forall_{\mathcal{L}} x:\tau \in x.\{c\} \\
 & | & \forall_{\mathcal{A}} x:\tau \in x.\{c\} \\
 & | & \exists_{\mathcal{L}} x:\tau \in x.\{c\} \\
 & | & \exists_{\mathcal{A}} x:\tau \in x.\{c\} \\
 & & \\
e_v & ::= & t \mid t.\mathcal{F}_v(t^*) \\
 & & \\
e_b & ::= & \neg e_b \mid f(t+) \\
 & | & t.\mathcal{F}_b(t^*) \mid x:\tau \leftarrow e_v
\end{array}
$$

Figure 3.4: The FJCOP language grammar.

These Datalog$^\neg$ goals are used to find AST nodes of type τ that do not satisfy a rule f. If execution of a Datalog$^\neg$ program augmented with a goal returns a non-empty set, then this indicates the FJCOP program would generate at least one error for the given AST node and the given FJCOP rule. If such an execution returns an empty set, then the FJCOP program would not generate an error from that rule. For example, the goal associated with the rule translated in Listing 3.15 would be

```
? :- type(X,JCAssign),¬(checkUntainted(X)).
```

To generate valid Datalog$^\neg$ programs, FJCOP must adopt the strategy of stratified predicates with respect to negation. Ceri et al. [CGT90] present an algorithm to determine stratification of a program. Application of this algorithm to the translated Datalog$^\neg$ program can determine whether the FJCOP program was originally stratified.

3.3.3 FJCOP Grammar

FJCOP is a subset of the actual JAVACOP language. This core subset of the JAVACOP language has been shown to be useful for a variety of non-trivial domain-specific and general purpose type systems including forms of confined type, ownership types, information flow systems, and design pattern checking [ANM06].

x, ℓ, and τ represent FJCOP variable names, Java literals, and Java type names, respectively. $\mathcal{F}_v$ and $\mathcal{F}_b$ represent functions from the FJCOP API which return non-`boolean` results and `boolean` results, respectively. Since the translation scheme presented here has different judgments for `boolean`-valued and non-`boolean`-valued expressions, this distinction is important. f is the set of predicate and rule names defined in the FJCOP program. Without loss of generality, this grammar assumes a canonical form wherein all parameters to a predicate or FJCOP API function are only variables or literals. A FJCOP program consists of one or more d elements.

3.3.4 Translation Judgment Notation

The Φ context found in many of the translation judgments represents the current scope's mapping from FJCOP variables to Datalog$^\neg$ variables; in other words, Φ is a function from FJCOP variables to a Datalog$^\neg$ variables. This is a one-to-one mapping, although the translation scheme does not strictly enforce one-to-oneness (nor does it enforce that Φ remain a function in the face of variable shadowing). A fresh Φ environment is generated for each FJCOP rule and predicate, thereby allowing local name reuse between top-level entities in FJCOP.

FJCOP API functions are redefined over a flattened AST in Datalog$^\neg$ to fit a logic programming style. For example, the FJCOP call to retrieve the right-hand-side of an assignment statement, `a.rhs()`, would correspond to the Datalog$^\neg$ expression $rhs(\mathtt{A},\mathtt{X})$

(as shown in Listing 3.15 where X corresponds to the result of the FJCOP expression.

The translation also uses a number of predicates which are not available from the FJCOP API, but which an implementation of FJCOP in Datalog$^\neg$ would supply. The *type* predicate indicates if an object has a particular Java type; the *object* predicate is defined for every object in the AST (which is used to restrict the range of variables used in negative literals); the *cons* and *nil* predicates define list relationships; and the *subnodes* predicate defines a traversal ordering of an AST subtree as a list of those nodes.

3.3.5 Translation Scheme

In this section, I present a subset of the judgments used in the translation scheme from FJCOP to Datalog$^\neg$. The full set of translation judgments can be found in Appendix A.

3.3.5.1 Value Expressions $\Rightarrow^v$

A translation rule of the form

$$\Phi \vdash e_v \Rightarrow^v E \mid T \dashv \Phi'$$

can be read as: Given the variable mapping context Φ, the FJCOP value expression e_v translates to the Datalog$^\neg$ term or literal E with Datalog$^\neg$ term T that will hold the result value that satisfies E when evaluated. The translation also updates the mapping context as Φ'. With value expressions, it is often important to refer back to the results of expressions in future evaluation. Thus, the translation rules for these kinds of expression result in both a term or literal, which will be used to build a rule in the Datalog$^\neg$ program, but also the term used to refer to the result of the evaluation for further transformation.

[TRANS-CONSTANT]

$$\frac{}{\Phi \vdash \ell \Rightarrow^{v} \ell \mid \ell \dashv \Phi}$$

[TRANS-VAR-KNOWN]

$$\frac{\Phi(x) = X}{\Phi \vdash x \Rightarrow^{v} X \mid X \dashv \Phi}$$

[TRANS-VALUE-FN]

$$\frac{\Phi \vdash t \Rightarrow^{v} T \mid T \dashv \Phi_0 \qquad \forall_{i=1}^{|\bar{t}|}.\Phi_{i-1} \vdash t_i \Rightarrow^{v} T_i \mid T_i \dashv \Phi_i \qquad X \text{ fresh}}{\Phi \vdash t.\mathcal{F}_v(\bar{t}) \Rightarrow^{v} \mathcal{F}_v(T, \overline{T}, X) \mid X \dashv \Phi_{|\bar{t}|}}$$

Translating constants and variables as shown in the TRANS-CONSTANT and TRANS-VAR-KNOWN rules are straightforward. The rule for translating variables currently not bound to a Datalog$^{\neg}$ variable is similarly straightforward and can be seen in Appendix A. The TRANS-VALUE-FN rule shown here simply flattens JAVACOP API functions into Datalog$^{\neg}$ literals.

3.3.5.2 Boolean Expressions $\Rightarrow^{b}$

A translation rule of the form

$$\Phi \vdash e_b \Rightarrow^{b} \overline{E} \dashv \Phi'$$

can be read as: Given the variable mapping context Φ, the FJCOP boolean expression e_b translates to the set of Datalog$^{\neg}$ literals $\overline{E}$ and updates the mapping context as Φ'.

[TRANS-PRED-CALL]

$$\frac{\forall_{i=1}^{|\bar{t}|}.\Phi_{i-1} \vdash e_i \Rightarrow^{v} T_i \mid T_i \dashv \Phi_i}{\Phi_0 \vdash f(\bar{t}) \Rightarrow^{b} f(\overline{T}) \dashv \Phi_{|\bar{t}|}}$$

[TRANS-BINDING-TYPE-TEST-1]

$$\frac{\Phi \vdash e_v \Rightarrow^v E \mid T \dashv \Phi' \qquad E \neq T}{\Phi \vdash x : \tau \leftarrow e_v \Rightarrow^b E, \texttt{type}(T,\tau) \dashv \Phi' \cup \{(x,T)\}}$$

[TRANS-BOOL-FN]

$$\frac{\Phi \vdash t \Rightarrow^v \Phi_0 \vdash T \mid T \qquad \forall_{i=1}^{|\bar{t}|}.\Phi_{i-1} \vdash t_i \Rightarrow^v T_i \mid T_i \dashv \Phi_i}{\Phi \vdash t.\mathcal{F}_b(\bar{t}) \Rightarrow^b \mathcal{F}_b(T,\overline{T}) \dashv \Phi_{|\bar{t}|}}$$

Rules TRANS-PRED-CALL and TRANS-BOOL-FN are straightforward reworkings of the TRANS-VALUE-FN rule from the previous subsection. Rule TRANS-BINDING-TYPE-TEST-1 makes use of the `type` predicate defined in Section 3.3.4 to ensure that the type test succeeds in order for the rule to be satisfied. One note of clarification regarding TRANS-BINDING-TYPE-TEST-1, the test for $E \neq T$ restricts this translation rule to translating a value expression that translates into a Datalog$^\neg$ literal and not in a Datalog$^\neg$ term. The other case can be found in Appendix A.

3.3.5.3 Constraints $\Rightarrow^c$

A translation rule of the form

$$\Phi \vdash c \Rightarrow^c \overline{E} \mid \overline{C}$$

can be read as: Given the variable mapping context Φ, the FJCOP constraint c translates to a set of Datalog$^\neg$ literals $\overline{E}$ and a set of Datalog$^\neg$ rules $\overline{C}$.

[TRANS-SEQUENCE]

$$\frac{\Phi \vdash c_1 \Rightarrow^c \overline{E_1} \mid \overline{C_1} \qquad \Phi \vdash c_2 \Rightarrow^c \overline{E_2} \mid \overline{C_2}}{\Phi \vdash c_1; c_2 \Rightarrow^c \overline{E_1}, \overline{E_2} \mid \overline{C_1} \cup \overline{C_2}}$$

[TRANS-WHERE-1]

$$\frac{F \text{ fresh} \qquad \overline{T} = \text{range}(\Phi) \qquad \Phi \vdash e_b \Rightarrow^b E \dashv \Phi' \qquad \Phi' \vdash c \Rightarrow^c \overline{E_c} \mid \overline{C}}{\Phi \vdash \textbf{where}(e_b)\{c\} \Rightarrow^c F(\overline{T}) \mid \left\{ \begin{array}{l} F(\overline{T}) :- E, \overline{E_c}. \\ F(\overline{T}) :- \bigwedge_{i=1}^{|\overline{T}|} \texttt{object}(T_i), \neg(E). \end{array} \right\} \cup \overline{C}}$$

[TRANS-FORALL-LIST]

$$\frac{\begin{array}{c} F, X_1, L, L' \text{ fresh} \\ \Phi \vdash x_2 \Rightarrow^v T_2 \mid T_2 \dashv \Phi \qquad \overline{T} = \text{range}(\Phi) \qquad \Phi \cup \{(x_1, X_1)\} \vdash c \Rightarrow^c \overline{E} \mid \overline{C} \end{array}}{\begin{array}{c} \Phi \vdash \forall_{\angle} x_1 : \tau \in x_2.\{c\} \Rightarrow^c \\ F(\overline{T}, T_2) \mid \left\{ \begin{array}{l} F(\overline{T}, L) :- \texttt{cons}(X_1, L', L), \texttt{type}(X_1, \tau), \overline{E}, F(\overline{T}, L'). \\ F(\overline{T}, L) :- \texttt{cons}(X_1, L', L), \neg(\texttt{type}(X_1, \tau)), F(\overline{T}, L'). \\ F(\overline{T}, L) :- \texttt{nil}(L). \end{array} \right\} \cup \overline{C} \end{array}}$$

[TRANS-EXISTS-LIST]

$$\frac{\begin{array}{c} F, X_1, L, L' \text{ fresh} \\ \Phi \vdash x_2 \Rightarrow^v T_2 \mid T_2 \dashv \Phi \qquad \overline{T} = \text{range}(\Phi) \qquad \Phi \cup \{(x_1, X_1)\} \vdash c \Rightarrow^c \overline{E} \mid \overline{C} \end{array}}{\begin{array}{c} \Phi \vdash \exists_{\angle} x_1 : \tau \in x_2.\{c\} \Rightarrow^c \\ F(\overline{T}, T_2) \mid \left\{ \begin{array}{l} F(\overline{T}, L) :- \texttt{cons}(X_1, L', L), \texttt{type}(X_1, \tau), \overline{E}. \\ F(\overline{T}, L) :- \texttt{cons}(X_1, L', L), F(\overline{T}, L'). \end{array} \right\} \cup \overline{C} \end{array}}$$

The first rule here, TRANS-SEQUENCE does a straightforward translation of a sequence of FJCOP constraints as the union (i.e., conjunction) of their respective translation elements. In Rule TRANS-WHERE-1, the `where` constraint is translated into two rules defining a new auxiliary predicate F which either satisfies both the constraint guard (E) and the body of the constraint ($\overline{E_c}$) or does not satisfy the guard. When negating the guard expression as a literal in Datalog$^\neg$, it is necessary to constrain the clause by confining the possible objects to the known set of objects via the `object` predicate. Otherwise, Datalog$^\neg$ will define the predicate over all constants that are not specifically defined in the program. Note that the TRANS-WHERE-1 rule handles the

case when the boolean expression translates into a single Datalog$^\neg$ literal. The two literal case is shown in Appendix A.

The Rules TRANS-FORALL-LIST and TRANS-EXISTS-LIST deal with quantification over lists. The list is flattened using the `cons` and `nil` predicates. Unlike in Rule TRANS-WHERE-1, constraining the negation in TRANS-FORALL-LIST does not require use of the `object` predicate as the `cons` predicate already limits the domain of the constants for the predicate F.

3.3.5.4 Predicate and Rule Definitions $\Rightarrow^d$

A translation rule of the form

$$d \Rightarrow^d \overline{C} \mid G$$

can be read as: The FJCOP definition d translates to the Datalog$^\neg$ clauses $\overline{C}$, and corresponds to the Datalog$^\neg$ goal G.

[TRANS-PREDICATE]

$$\frac{\Phi_0 = \emptyset \qquad \forall_{i=1}^{|\overline{x}|}.\Phi_{i-1} \vdash x_i \Rightarrow^v X_i \mid X_i \dashv \Phi_i \qquad \Phi_{|\overline{x}|} \vdash c \Rightarrow^c \overline{E} \mid \overline{C}}{\textbf{declare}\ f(\overline{x} : \overline{\tau})\{c\} \Rightarrow^d \{f(\overline{X}) :- \bigwedge_{i=1}^{|\overline{x}|} \texttt{type}(X_i, \tau_i), \overline{E}.\} \cup \overline{C} \mid \emptyset}$$

[TRANS-RULE]

$$\frac{\emptyset \vdash x \Rightarrow^v X \mid X \dashv \Phi \qquad \Phi \vdash c \Rightarrow^c \overline{E} \mid \overline{C}}{\textbf{rule}\ f(x : \tau)\{c\} \Rightarrow^d \{f(X) :- \texttt{type}(X, \tau), \overline{E}.\} \cup \overline{C} \mid \{\texttt{?} :- \texttt{type}(X, \tau), \texttt{not}(f(X)).\}}$$

As is expected, both `declare` and `rule` constructs result in the definition of corresponding predicates in Datalog$^\neg$. The predicate definitions consist of the translated constraints as well as type tests for the declared parameters.The `rule` translation also produces a goal of the form $\texttt{?} :- \texttt{type}(X, \tau), \texttt{not}(f(X))$. which, when added to a

Datalog$^\neg$ program, will determine set of AST nodes of type τ that do not satisfy the rule f, therefore point to a violation of the FJCOP type system.

3.3.5.5 Program $\Rightarrow^p$

A translation rule of the form

$$\overline{d} \Rightarrow^p \overline{C} \mid \overline{G}$$

can be read as: The FJCOP program defined via definitions $\overline{d}$ translates to the Datalog$^\neg$ program $\overline{C}$ and a set of goals $\overline{G}$.

$$\frac{\text{[TRANS-PROGRAM]} \quad \forall_{i=1}^{|\overline{d}|}.d_i \Rightarrow^d \overline{C_i} \mid G_i}{\overline{d} \Rightarrow^p \bigcup_{i=1}^{|\overline{d}|} \overline{C_i} \mid \bigcup_{i=1}^{|\overline{d}|} G_i}$$

As expected, the transformation of a FJCOP program is just the union (i.e., conjunction) of all the transformed `rule` and `declare` constructs.

3.4 The JavaCOP Dataflow Framework

Some programming disciplines require flow-sensitive reasoning. For example, a programmer should never dereference an object unless it can be guaranteed *at that program point* that the object is not null. A programmer might accomplish this by placing an `if` statement checking for nullity around the dereference. Listing 3.16 illustrates code that a practical, usable *non-null* discipline checker should allow. The JAVACOP language can handle simple forms of flow-sensitivity by pattern matching against large portions of code–such as in the case of an `if` statement–but pattern matching will not work for the general case of flow-sensitivity.

```
class Person {
  @NonNull String name = "Chris";

  void setName(String newname) {
    if (newname == null)
      name = "";
    else
      name = newname;
  }
}
```

Listing 3.16: Example code that requires flow-sensitive reasoning.

The need for flow-sensitive reasoning in building sound, expressive type systems has been demonstrated in recent literature (e.g., [FL03, AKC02, Boy01]). The following approach to providing users with a practical way to write these kinds of checkers was adopted in JAVACOP:

1. JAVACOP provides a generic dataflow framework that allows users to easily define analyses that determine facts that hold at each program point.

2. JAVACOP performs user-defined flow analyses prior to enforcing a checker's core JAVACOP rules.

3. Results of the analyses are made available to the core JAVACOP rules via decorations on the AST nodes allowing users to easily write checkers that accept or reject programs based on flow-sensitive information.

This section describes JAVACOP's support for defining checkers that involve flow-sensitive reasoning. To illustrate how checkers can make use of this information, I explain the framework in terms of a *non-null* discipline checker with the goal of showing how to allow the program in Listing 3.16 to successfully type check. A non-null type system ensures that no null values are stored in references that are marked as non-null. Non-null references are important because dereferencing them is always safe:

doing so will never cause unexpected null pointer exceptions at runtime. The basic rules of a non-null checker are very similar to those I previously defined for untainted in Listings 3.2, 3.4, 3.7, and 3.9. A more detailed example of the non-null checker is described in Section 3.7.

3.4.1 Specifying Dataflow Analyses

JAVACOP's flow framework allows users to easily create forward, gen/kill style, intraprocedural dataflow analyses. In general, describing an analysis of this type requires specifying:

1. the dataflow facts being tracked (e.g., definition sites for a reachable definitions analysis or program expressions for an available expressions analysis)
2. the sets of dataflow facts that are generated by program expressions
3. the sets of dataflow facts that are killed by program expressions
4. how to combine sets of dataflow facts at control flow merges

In the JAVACOP framework, this is accomplished by defining a Java class that implements an interface described below. In contrast to the declarative language for writing type systems, JAVACOP gives users the full expressive power of Java to specify analyses. In particular, this allows the user to choose a representation for sets of dataflow facts that is appropriate for the analysis (e.g. a bit vector versus a hash map). In the future, the JAVACOP language may be extended to provide a declarative interface to the dataflow framework (e.g., in the manner of bddbddb [WL04]). Such an extension may fit nicely with the JAVACOP semantics given in Section 3.3.

To define a dataflow analysis in JAVACOP, the user provides a Java class that represents a set of dataflow facts and that implements the `FlowFacts` interface shown

```
interface FlowFacts {
  FlowFacts genSet(JCTree node);
  FlowFacts killSet(JCTree node);

  //Allow branch-condition-sensitive analysis
  FlowFacts genSetTrue(JCTree node);
  FlowFacts killSetTrue(JCTree node);
  FlowFacts genSetFalse(JCTree node);
  FlowFacts killSetFalse(JCTree node);

  //Operations for combining sets of FlowFacts
  FlowFacts addSet(FlowFacts f);
  FlowFacts removeSet(FlowFacts f);
  FlowFacts meetWith(FlowFacts f);
}
```

Listing 3.17: The interface for defining dataflow analyses.

in Listing 3.17. The user-defined `genSet` and `killSet` functions determine how to change an incoming set of dataflow facts as a result of traversing the given expression. The `addSet` and `removeSet` functions respectively are used to update the current set of facts with the gen and kill sets, and the `meetWith` function is used to combine sets at control flow merges. Note that the user-defined `genSet` and `killSet` functions are consulted for individual program expressions, not for control flow statements such as `if`/`else`, `try`/`catch`/`finally`, loops, and labeled breaks. These statements are handled by the framework's traversal code, shielding users from the complexities of Java's control flow. For simple cases of path-sensitivity, such as distinguishing between the then-branch and an else-branch of an if statement, the API provides the methods `genSetTrue`, `genSetFalse`, `killSetTrue`, and `killSetFalse`.

3.4.2 An Example Analysis for the Non-null Checker

Listing 3.18 shows the `FlowFacts` implementation for an analysis that can determine that `newname` is non-null when assigned to `name` in Listing 3.16. The `genSetFalse` implementation checks if the given expression has the form `"localVar == null"` . If so, it returns a set containing the fact that `localVar` is non-null, since this clearly holds when the expression evaluates to false. This fact might be invalidated by a reassignment to `localVar`. This is handled by the `killSet` implementation. The implementations for the other gen and kill functions required by the `FlowFacts` interface are not shown and simply return an empty set of dataflow facts. The implementation distributed as part of the JAVACOP suite enhances this one by allowing `null` to appear on the left of an equality test and allowing for `!=` tests. The implementation additionally generates non-null facts on certain assignments to local variables that are clearly not null, such as string literals and (boxed) primitive literals.

3.4.3 Accessing Analysis Results from JavaCOP Rules

When using dataflow analysis, each expression node in the AST is decorated with the set of facts valid before its evaluation. The JAVACOP rules can then query these decorations in order to incorporate flow-sensitive reasoning. Listing 3.19 shows the JAVACOP code that must be added to the non-null checker in order for JAVACOP to accept the code in Listing 3.16. The `flowfact` declaration allows the user to specify that a particular implementation of the `FlowFacts` interface should be used during the analysis pass. A JAVACOP file may indicate that several analyses should be performed by including multiple `flowfact` declarations.

Incorporating flow sensitivity into a non-null checker only requires adding one new case to the `definitelyNotNull` predicate (similar to the one for untainted shown in

```
class NonNullFacts implements FlowFacts {

  HashSet<Symbol> nonnulls = new HashSet();

  //Explicit non-null test generates non-null fact
  FlowFacts genSetFalse(JCTree node){
    NonNullFacts gen = new NonNullFacts();
    if (node instanceof JCBinary
        && node.tag == JCTree.EQ) {
      JCBinary b = (JCBinary)node;
      Symbol s = getSymbol(b.lhs);
      if (b.rhs.getKind() == Kind.NULL_LITERAL
          && isLocal(s))
        gen.nonnulls.add(s);
    }
    return gen;
  }

  //Re-assignment kills non-null fact for local variable
  FlowFacts killSet(JCTree node){
    NonNullFacts kill = new NonNullFacts();
    if (node instanceof JCAssign){
      Symbol s = getSymbol(((JCAssign)node).lhs);
      if (isLocal(s))
        kill.nonnulls.add(s);
    }
    return kill;
  }

  //Define the meet operation as intersection
  FlowFacts meetWith(FlowFacts f){
    nonnulls.retainAll(((NonNullFacts)f).nonnulls);
    return this;
  }

  //Rules can use this method to get non-null facts
  boolean isIdentNonnull(JCIdent id){
    return nonnulls.contains(getSymbol(id));
  }
}
```

Listing 3.18: FlowFacts for non-null analysis.

```
flowfact nonnull.NonNullFacts;

declare definitelyNotNull(JCIdent id){
  require(NonNullFacts f;
          f <- id.getFlowFacts("nonnull.NonNullFacts")){
    require(f#isIdentNonnull(id));
}}
```

Listing 3.19: A JAVACOP declaration using non-null dataflow facts.

Listing 3.9), as shown in Listing 3.19. The case queries the `NonNullFacts` object that decorates an identifier's AST node in order to check whether or not the dataflow analysis determined the identifier to be non-null at this point. Note that the `FlowFacts` interface itself does not provide a means of accessing the dataflow facts. The user is free to provide whatever methods are appropriate and convenient for querying a particular implementation. In this example, the `isIdentNonnull` method from Listing 3.18 serves this purpose[1].

3.4.4 Flow Analysis Implementation

The goal of implementing the JAVACOP dataflow framework was to make it easy for users to specify the kinds of analyses likely to be needed when building pluggable type checkers – analyses similar to those required by the existing Java type system. The `javac` compiler already performs a flow analysis in order to flag errors for uninitialized local variables and final fields, unreachable code, and uncaught exceptions. This well-tested, efficient code was generalized in JAVACOP to make calls to the `FlowFacts` interface during traversal of the AST.

It is worth noting that although JAVACOP dataflow analyses operate on the AST (as opposed to a control flow graph), users of the framework need not be concerned

[1] Recall that the # operator allows access to Java methods that are not part of the JAVACOP API.

with the complexities of Java's control flow. This is all handled by the framework's traversal code. For example, the analysis defined by the simple `NonNullFacts` class in Listing 3.18 not only handles the if-statement case from the motivating example in Listing 3.16, but also correctly derives the dataflow information over all similar constructs such as loop conditionals.

The decision to adapt the existing OpenJDK analysis imposes some limitations on the analyses that JAVACOP supports. Only forward, gen/kill dataflow analyses are permitted. But in practice, the flowfacts framework is able to handle many useful analyses. These analyses were built with minimal per-analysis implementation effort; an average of 115 non-comment, non-whitespace lines of code sufficed for each.

3.5 Type System Testing

Just as Java programmers make mistakes in their programs, JAVACOP users may introduce errors in their pluggable type systems. In this section we describe a practical approach to giving JAVACOP users confidence in the correctness of their type systems. In Section 3.5.1 we describe the two-stage testing approach supported by our framework and in Section 3.5.2 we report on our experience using the test harness.

3.5.1 Two-Stage Testing Approach

The first stage, the *compilation* stage, of our testing framework leverages the fact that type system developers will have an understanding of the kinds of programs that should and should not successfully type check, much as unit test developers have a general idea of what the results of their unit tests should be. Verifying that type checkers generate the expected compilation results requires three components from the developer: the JAVACOP type system being tested, a test suite of Java programs, and the expected

```
public abstract class RawTestParent {

        public RawTestParent() {
                m();
        }

        @RawThis abstract protected void m();

}

public class RawTest extends RawTestParent {
        public @Nonnull Integer f1 = new Integer(3);
        public @Nonnull Integer f2 = new Integer(4);

        public RawTest() {
                super();
        }

        @RawThis protected void m() {
                f1 = f2 * 2;
        }
}
```

Listing 3.20: A couple of Java classes defining a test from the non-null test suite. This test demonstrates a case where a potentially uninitialized `@NonNull` field could result in a null dereference during construction.

outcome of compiling each test program with the JAVACOP type system. Each of the programs in the test suite is then compiled using the specified JAVACOP type system and the results are compared with the expected outcome.

An example from the non-null test suite is shown in Listing 3.20. This examples demonstrates how virtual method dispatch, field initialization, and non-null references interact during object initialization. At first glace, it may seem safe to do the assignment from `@Nonnull` field `f2` to `@Nonnull` field `f1` in method `m` of the class `RawTest`. However, `f2` may not have been initialized yet if `m` is called from the superclass's (`RawTestParent`'s) constructor. Thus, this test is expected to *fail* with respect to the

```
Loading constraints class: javacop.nonnull.Nonnull
Done, loaded 1 constraint sets.
javacop/runtime/test/examples2/RawTest.java:14: Nonnull:
Possibly null value where a @Nonnull value is expected
                f1 = f2;
                     ^
1 error
```

Listing 3.21: The expected result of running the test shown in Listing 3.20.

non-null type system. JAVACOP stores the expected outcome as a text file containing the expected output of the compiler. For the test in Listing 3.20, the expected outcome is shown in Listing 3.21.

The compilation stage of the test framework allows JAVACOP users to test whether the intended static programming discipline is being implemented correctly by the rules. However, these tests do not provide feedback on whether the type system ensures *intended* program properties, only the *expected* results of type checking. The distinction between intended and expected here is subtle: the developer of a test suite expects the tests to provide certain type checking results, but may have nonetheless created a test which she believes should type check but actually violates the intended runtime invariants associated with the type system.

For example, consider again the non-null type system. A field or variable declared `@NonNull` is intended to never have the value `null` at runtime, thereby preventing null dereferences. However, suppose the developer had mistakenly denoted that the test in Listing 3.20 should *pass* type checking. A type system that does not flag this test as an error would successfully meet the expectations of the type system according to the developer, but fail to ensure the intended non-null properties.

The second stage (the *runtime* stage), of JAVACOP's testing framework allows users to test whether these *intentions* are violated for programs that their type system

accepts. I allow JAVACOP users to express the intended runtime properties with a simple API that supports runtime instrumentation of Java bytecode with user-defined checks. This approach offers an advantage over formal specification of an invariant in that the invariant can be tested directly on the objects at runtime. Thus, instead of having to reason about how Java deals with heaps, multithreading, dispatch, etc., developers can write simple code that examines the objects available at runtime.

With this instrumentation facility, the user optionally specifies a Java runtime-check method for each type of bytecode instruction to be executed immediately before all instructions of that kind. This method can check properties of the bytecode instruction and signal an error when an invariant is violated. If such an error is signaled when executing a program that was accepted by a type checker, then a hole in that type system is exposed. Using bytecode instructions as the entry point for runtime tests is a simple solution, but it does pose one problem. Java programmers to do not always know which bytecode instructions correspond to the source code elements. This could potentially be handled by adding more test entry points to the JAVACOP instrumentation facility such as at all method invocations, all dereferences, all field updates, etc. While JAVACOP does not currently support this, it is something I am considering adding in the future.

Listing 3.22 shows an excerpt from the runtime checker for non-null properties. The excerpt defines three methods: `test_aux`, `testPutField`, and `testGetField`. Both of the methods `testPutField` and `testGetField` define define how their respective bytecode instructions should be instrumented. The arguments to the methods are references that are copied off of the virtual machine stack to ensure unintended effects to these objects are minimized. Metadata annotations on each formal describe the what each argument is meant to represent for that instruction.

Each instrumented bytecode instruction has its own set of parameters that can be

accessed[2]. The `testPutStatic` method the type of the object whose field is being updated (`@ReceiverObject recv`), the name of the field (`@FieldName name`), and the object being assigned to the field (`@ActionObject o`) are all provided. In comparison, the `testGetField` method provides the actual object of the receiver instead of the type of class `@ReceiverObject recv` and lacks the `@ActionObject` parameter. As the method `test_aux` does not correspond to any particular bytecode instruction, it will result in no additional automated instrumentation. However, the instrumentation methods use `test_aux` for code reuse. The `generateError` method generates a JAVACOP runtime exception from the error message it is passed and should be called by all test methods. It will only generate errors if it is given a non-null string.

The test framework incorporates the user-defined runtime invariant tests as follows. Runtime instrumentation classes, such as the one shown in Listing 3.22, are used to build an adapter that instruments bytecode dynamically when a class is loaded by the JVM. I implemented this on top of the ASM bytecode rewriting framework [BLC02]. Each program in the test suite that is accepted by the user's JAVACOP checker in the first stage of testing continues to the second stage, where it is compiled and executed using the instrumenting adapter. Any violations are reported to the user and indicate that a program satisfying the type system's discipline nevertheless fails to meet the desired runtime invariants.

Listing 3.23 shows, at the source code level, the result of instrumenting the test from Listing 3.20 via the non-null runtime instrumenter from Listing 3.22. Running this instrumented program will result in an exception being generated before the assignment from `f2` to `f1` in `RawTest`'s method `m` when called from the constructor of `RawTestParent`. In practice, this instrumentation is done only at the level of the bytecode when the class is loaded, but is shown at the source code level for clarity and

[2] All test methods actually share two common parameters, the object in whose context the instruction is executed and that object's type, but these parameters are omitted for brevity.

```
public class NonnullTestMethod {

  /* common nullity test and message abstraction */
  private static String test_aux(Object o) {
    return (o == null)?"Dereference of null object"
                      :null;
  }

 /* check for two properties:             *
  * 1. null assigned into @Nonnull field *
  * 2. dereferenced object is null       */
  public static void testPutStatic(
      @ReceiverObject Class recv,
      @FieldName String name,
      @ActionObject Object o){
    String message = null;
    try{
      if(recv.getField(name).hasAnnotation(Nonnull.class)
          && o == null)
        message = "Cannot assign null to field "+name;
    } catch (Exception e) {
      /* exception cannot be generated */
    }
     generateError(message);
  }

  /* the other tests all test for receiver nullity */
  public static void testGetField(
      @ReceiverObject Object recv,
      @FieldName String name) {
    generateError(test_aux(recv));
  }
 ⋮
}
```

Listing 3.22: An excerpt from the Java class that specifies the runtime semantics for a `@NonNull` type system. This excerpt signals an error when a null value is assigned into a `@NonNull` field at runtime.

brevity.

3.5.2 Experience

Runtime Instrumentation I implemented four non-trivial runtime instrumenters for use with the test framework. The non-null checker is 36 LOC, instrumenting four bytecode instructions. It replicates the behavior of the Java virtual machine's nullity checking and extends this to include a test to ensure a `null` value is never stored in a `@NonNull` reference. The *confined* type system [VB99] instrumenter consists of 37 LOC and instruments seven bytecode instructions. Confined type systems are meant to ensure that confined objects are encapsulated by their package, so the instrumenter checks for illegal dereferences and assignments of those objects outside of their packages. I also implemented runtime instrumenters for a *race-condition detection* type system [FF00] (43 LOC, four bytecode instructions), which determines if a field is accessed without its lock being held, and for our implementation of Java's `final` class modifier (17 LOC, one bytecode instruction), which checks if a subclass of a `@Final` type is ever instantiated at runtime.

3.6 Domain-Specific Checkers: Polyglot & SCJ

This section illustrates how JAVACOP can be used to enforce domain-specific properties on Java projects. These kinds of domain-specific checkers point to the core motivation for the work presented in this dissertation. Building such checkers from scratch requires a non-trivial amount of time and a grasp on the fundamentals of Java compiler architecture. By abstracting away the details of the compiler, the disciplines can be developed quickly and maintain a close correspondence with their intent.

Although I have used JAVACOP to write a number of these pluggable type systems,

```
public class RawTest extends RawTestParent {

  public @Nonnull Integer f1;
  public @Nonnull Integer f2;

  public RawTest() {
    super();
    if (this == null)
      throw new JavaCOPException(...);
    Integer temp = new Integer(3);
    if (temp == null)
      throw new JavaCOPException(...);
    f1 = temp;
    temp = new Integer(4);
    if (temp == null)
      throw new JavaCOPException(...);
    f2 = temp;
  }

  @RawThis protected void m() {
    if (this == null)
      throw new JavaCOPException(...);
    if (f2 == null)
      throw new JavaCOPException(...); // throws exception
    f1 = f2;
  }
}
```

Listing 3.23: An updated version of the `RawTest` classes from Listing 3.20 that shows how runtime instrumentation would affect the code. Instrumentation actually occurs at the bytecode level when the class is loaded.

I chose to examine two particularly compelling domain-specific examples here. One of the reasons I find these examples so compelling is how they show even simple project-specific disciplines can be useful for building better software. The first checker is for the Polyglot extensible compiler framework from Cornell [NCM03]. This checker determines whether projects based on Polyglot appropriately use the design patterns which provide Polyglot with its extensible behavior. The second example is a checker for Safety Critical Java Technology from the Java Specification Request 302 team. The discipline of this checker is very straightforward, but any mistakes in implementing the discipline can cause program errors that result in catastrophic physical damage. Thus, it is vitally important to follow the discipline to create safe programs.

There are a number of other domain specific disciplines that I have implemented which are not discussed here but can be found in the JAVACOP papers [ANM06, MME]. These disciplines include a very large checker for Enterprise JavaBeans 3.0 conformance [DeM04], a *degenerate classes* micropatterns detector [GM05], PMD Java style checkers [Cop05], and a checker for determining whether JUnit tests [Jt00] provide useful feedback.

3.6.1 Design Patterns in Polyglot

Polyglot [NCM03] is an extensible compiler framework for Java from Cornell, written in Java. Polyglot has been publicly available since 2004 and used by many researchers to implement Java language extensions [Pt04]. Polyglot employs a number of design patterns that are not checked by the standard Java type system, so programmers must manually ensure that their code conforms. The Polyglot type system in JAVACOP tests for proper adherence to the following idioms:

1. Polyglot employs the factory design pattern [GHJ95] for both AST nodes and for “type objects” that hold the type information about a class. The checker

requires that any expression of the form `new C(...)`, where `C` is a subtype of the `Node` interface, appear only in classes that subtype `NodeFactory`, and similarly for type objects.

2. Each AST node in Polyglot is represented by both a class and an interface. The intent is that clients of a Polyglot extension should only manipulate AST nodes through their associated interfaces. The checker requires that a node class is never used as the type of a public or package-level field or as the argument or result type of a public or package-level method.

3. Polyglot uses a variant of the visitor design pattern [GHJ95] to allow implementers to traverse the AST. Each node class must have a `visitChildren` method that implements the traversal behavior for that kind of node. The checker requires that each node class overrides the `visitChildren` method if it adds at least one new field of type `Node` (or a subtype).

4. Polyglot employs a notion of *delegates* [NCM03] that allows the behavior of an AST node to be modified modularly without requiring the creation of a subclass. Each AST node has a pointer to a delegate object, and clients should always invoke certain operations of a node (defined in the `NodeOps` interface) through the node's delegate (e.g., `n.del().typecheck()` instead of simply `n.typecheck()`). The checker enforces this rule.

The entire Polyglot checker consists of 80 lines of (non-blank, non-comment) JAVACOP rules and auxiliary predicates. JAVACOP's declarative nature makes each rule relatively straightforward to understand. For example, Listing 3.24 enforces the factory design pattern for AST nodes. The rule directly corresponds to the English description provided above. The auxiliary user-defined predicate `isSubtype` checks whether a type (represented by its symbol) has a particular supertype.

```
rule nodeFactory(JCNewClass nc) {
  where(isSubtype(nc.constructor.owner,
                  "polyglot.ast.Node")) {
    require(isSubtype(env.enclClass,
                      "polyglot.ast.NodeFactory")):
      error(nc, "Nodes cannot be directly instantiated "+
                "outside of the node factory!");
}}
```

Listing 3.24: A JAVACOP rule enforcing Polyglot's factory design pattern.

Table 3.1: Results of running the Polyglot style checker.

Compiler	*LOC*	*Errors*	
		signaled	*actual*
Polyglot-1.3.4	20910	7	7
Polyglot-2.3.0	25154	7	7
Polyglot5	7800	3	3
JPred	3343	0	0
eJava	2458	2	2
jet-0.9.0	921	0	0
jif-3.1.1	22020	14	12

The Polyglot style checker was then run on seven Polyglot compilers/extensions; the results are shown in Table 3.1. The first two compilers are the Polyglot 1.x and 2.x branches from Cornell, respectively. The next three compilers in the table are Polyglot extensions from UCLA to respectively support Java 1.5 features [Pt07], predicate dispatch [Mil04], and expanders [WSM06]. The last two compilers are Polyglot extensions from Cornell to respectively support nested intersection [NQM06] and secure information flow [Mye99].

The second column in the table lists the (non-blank, non-comment) lines of Java code in the compiler or extension, ignoring generated code (e.g., from the parser generator) and other special files. For each compiler, the number of errors signaled by the checker is listed. These error messages were provided to the developers of the compilers and the developers were asked to verify which were (in their opinion) actual errors. As the last column in the table shows, of the 33 errors signaled across all compilers, 31 of them were actual errors.

All 14 errors in the two Polyglot base compilers were violations of the fourth idiom described above, related to delegates. These 14 errors represent 12 distinct errors: two errors from the 1.3.4 version were duplicated in the 2.3.0 version. All of these errors have been fixed by the developer in the latest releases of the 1.x and 2.x Polyglot branches.

Nine of the 14 errors signaled for Jif were violations of the first idiom described above, related to factories. these nine errors were considered false positives by the developer. One involved a temporary node class used only during parsing, which was deliberately not given an associated factory method and so is always directly instantiated. The other pertained to a class that was being used as a factory for certain type objects, even though it was not a subtype of the standard Polyglot interface for type factories. A user could easily employ an annotation like `@TypeFactory`, along with a

```
rule nodeFactory(JCNewClass nc) {
  where(isSubtype(nc.constructor.owner,
                  "polyglot.ast.Node")) {
    require(isSubtype(env.enclClass,
                      "polyglot.ast.NodeFactory") ||
            env.enclClass.hasAnnotation("NodeFactory")):
      error(nc, "Nodes cannot be directly instantiated "+
                "outside of the node factory!");
}}
```

Listing 3.25: A JAVACOP rule enforcing an extension of Polyglot's factory design pattern.

simple modification to our JAVACOP rule, to eliminate this false positive. A modified version of the Polyglot rule from Figure 3.24 with this effect is shown in Figure 3.25.

3.6.2 Safety Critical Java (SCJ) Checker

Safety Critical Java (SCJ) [SCJ] is a subset of the Java language, along with a new set of dedicated Java services, that can be used for creating safety critical applications in Java. Such applications require that they must have good performance and reliability because of their use in situations where failure can result in damage to human life. As a result, numerous standards bodies have defined what it means for a program to be certified as safety critical. The SCJ team chooses to adhere closely to the definition of safety critical as required for certification using the Software Considerations in Airborne Systems and Equipment Certification (DO-178B) [RTC92] Standard. This is the same standard used by the US Federal Aviation Administration.

SCJ programs are meant to ease the process of certification under the DO-178B standard by reducing the complexity of the programs. The reduction in complexity is achieved through the definition of three conformity levels to safety critical standards defined in the SCJ specification document: levels 0, 1, and 2. Level 0 is used

for simpler applications that have low complexity and little difficulty meeting memory constraints and real-time deadlines. Levels 1 and 2 allow the use of more Java libraries and thus increase the complexity of the applications. Java virtual machines are similarly built to run at these levels to ensure that complexity is not added in at runtime.

Level information is introduced in the code via the use of the `@SCJAllowed` annotation. This annotation can take the name of the defining level as a metadata argument. Thus, a class declared at level 0 would be annotated with an `@SCJAllowed(Level_0)` annotation. For the Java virtual machine, the compliance level is specified as a command line argument.

The key to retaining the modularity and code reuse benefits of Java in SCJ is in allowing programs defined at lower levels to remain valid programs for the higher levels. Simple and universal classes, such as `Object`, would be defined at level 0, and thus be usable in any level's programs. However, a complicated class like `Thread` would be defined at a higher level to prevent its use in a lower level program. If a Java virtual machine is run at a particular level, then it will refuse to load a class defined at a higher level.

The SCJ JAVACOP type system enforces the following idioms:

1. The default annotation for an otherwise unannotated class is `@SCJAllowed(Level_0)`.

2. When a class is annotated with an `@SCJAllowed(Level_n)` annotation, its members assume a default annotation of `@SCJAllowed(Level_n)`. Members of a class may override the default behavior by declaring a higher level than their owning class.

3. All annotations must be preserved through subclassing and method overriding.

4. Nested non-static classes must be at least as restrictive as their enclosing classes. Nested static classes may be given any level of compliance.

5. Fields, methods, and constructors may only be accessed in a context with a level at least as high as the level with which they are annotated.

The JAVACOP implementation of this discipline consists of 65 (non-blank, non-comment) lines of code. There are 10 predicate declarations and 5 rules in the type system. Because the rules and predicates remain relatively small, they are easy to read and modify. Listing 3.26 shows a predicate and rule from this set. The rule codifies that fields declared in a class may only broaden the compliance level specified by its enclosing class. The predicate declaration defines the `allowedSubtype` predicate used in the rule. This predicate makes use of the ability to reconstruct annotations from declarations to directly compare the `Level`s of two entities.

The type system was used by the SCJ developers to test their spec implementation. This spec implementation consists of an annotated versions of the `java.lang` package and the `Vector` class from the `java.util` package from standard Java; an annotated version of the `javax.realtime` package from the Real-Time Specification for Java [BBD00]; and a proposed new package `javax.safetycritical`. In total, this amounts to just under 130 Java files.

Type checking these classes resulted in the detection of 29 errors related to SCJ in the spec implementation[3]. 17 of these errors related to missing annotations on classes and methods related to subtyping. 8 of the errors related to method and constructor invocations in a lower compliance level context. Listing 3.27 shows one such error. Lastly, 4 of the errors were a result of a subclass defining a higher level than their parent. Based on the detection of these 4 errors, it was recommended to change the

[3]The presence of other Java errors unrelated to SCJ may have resulted in under-reporting of actual errors.

```
/* Fields must be declared at the same or higher level of
 * compliance as their enclosing class.
 */
rule fieldDef(JCVariableDecl v){
  where(!v.isLocal && hasSCJAllowed(v)){
    where (hasSCJAllowed(env.enclClass)){
      require(allowedSubtype(env.enclClass.sym, v.sym))
        :error(v,...);
    }
  }
}

/* One declared entity is a subtype of another if it has a
 * lower declared @SCJAllowed level.
 */
declare allowedSubtype(Symbol x, Symbol y){
  require(SCJAllowed x_annot;
              x_annot <- x.getAnnotation("SCJAllowed")){
  require(SCJAllowed y_annot;
              y_annot <- y.getAnnotation("SCJAllowed")){
    require(Level x_value; x_value <- x_annot#value()){
    require(Level y_value; y_value <- y_annot#value()){
      require(x_value#compareTo(y_value) <= 0);
}}}}}
```

Listing 3.26: A rule and predicate declaration from the SCJ type system. The rule requires members declare the same level of or higher compliance as their enclosing class. The predicate defines a subtype relationship among `@SCJAllowed` annotations.

```
public abstract class EventHandler
  extends BoundAsyncEventHandler {
  ⋮
  private Runnable task_ = new Runnable() {
    public void run() {
      // LEVEL_1 method accessed in LEVEL_0 context!
      handleEvent();
    }
  };
  ⋮
  @SCJAllowed(LEVEL_1)
  public abstract void handleEvent();
  ⋮
}
```

Listing 3.27: A Java class demonstrating an error in the spec implementation of Safety Critical Java caught by the JAVACOP SCJ checker: the method `handleEvent()`, defined at `Level_1`, is used in a default context of `Level_0` in the anonymous `Runnable` instance assigned to field `task`.

SCJ specification to allow this kind of definition; the virtual machine of level *n* will not load a class defined at a level higher than *n*, so such *unsafe* subclasses cannot affect performance on a virtual machine running at the level of the parent.

Looking at the results points out that even domain experts adhering to a simple discipline can easily make mistakes in their code. While the 17 errors related to missing annotations were likely the results of incremental porting from the original Java versions, the 8 errors related to method and constructor invocations are more serious. It is unlikely that a developer will remember the compliance level of every method or constructor in a project, so these kinds of mistakes are bound to happen occasionally, even with expert programmers.

3.7 Advanced Type Checker: Non-null Types

This section demonstrates JAVACOP's ability to express state-of-the-art type systems by focusing on an in-depth description of a non-null checker for preventing null pointer dereferences in Section 3.7.1. Recent literature (e.g., [FL03, AKC02, Boy01]) has shown that type systems for these properties that are both sound and expressive enough to be usable in practice employ reasoning that is subtle and complex. By using JAVACOP's declarative rule language introduced in Section 3.2 in concert with the dataflow framework presented in Section 3.4, it is possible to build robust, usable checkers for these properties.

While not presented here, numerous other research type systems have been implemented in JAVACOP including confined types [VB99] as described in Section 1.1.1, scoped types [ZNV04], race condition detection types [FF00], island types for object encapsulation [Hog91], Javari-style reference immutability types [BE04, TE05], types for generic ownership in Java (OGJ) [PNC06], and unique reference types [AKC02, Boy01]. More details about these type systems can be found in prior JAVACOP publications [ANM06, MME].

3.7.1 Non-null Type System

As stated previously in Section 3.4, a non-null type system ensures that no null values are stored in references that are marked as non-null. Non-null references are important because dereferencing them is always safe: doing so will never cause unexpected null pointer exceptions at runtime. The basic rules of a non-null checker are very similar to those I previously defined for untainted in Listings 3.2, 3.4, 3.7, and 3.9. In particular, it is essential that *non-null* types be treated as a subtype of an unannotated type just as in the case for untainted. Using the JAVACOP dataflow framework, explicit runtime

checks in the source code can be used to ensure non-nullity of a reference before dereferencing it or storing it into a location annotated with `@NonNull`.

3.7.1.1 Adding Raw Types

Due to the semantics of Java object construction, it is impossible to guarantee that a field annotated with `@NonNull` *never* contains `null` – even if it is initialized at the declaration site. The code in Listing 3.28 shows how a field can be accessed before it is initialized. Since `A`'s constructor is executed before subclass `B`'s field initializers are run, the overridden `init` method in `B` will dereference field `f` before it is initialized. The Java runtime will have stored `null` in `f` and a null pointer exception will result.

As this example shows, it would not be sound for a non-null checker to simply enforce that a field annotated with `@NonNull` have a non-null initializer in its declaration. To handle this situation, my checker supports the *raw types* approach invented by others [FL03]. I assume that fields declared as non-null may in fact be null while the object is under construction, or *raw*. I introduce two new annotations, `@Raw` and `@RawThis`, that indicate that a method parameter or the receiver of a method call, respectively, may be under construction. JAVACOP rules are used to enforce that a constructor only invokes `@RawThis` methods on the object being constructed and only passes `this` to a method as a `@Raw` parameter. Listing 3.29 fixes the erroneous code from Listing 3.28 so that the checker accepts it. The type system requires `init` to have the `@RawThis` annotation, since it is called while the object is under construction. Once the method is marked as having a potentially raw receiver, the checker requires that a runtime nullity check is inserted before dereferencing field `f`, since it may contain `null` despite its `@NonNull` annotation.

Once the raw types mechanism is in place, it is overly restrictive to insist that a `@NonNull` field be initialized at its declaration site. It is sufficient to check that all

```
class A {

  A() {
    init();
  }

  void init() { }
}

class B extends A {

  @NonNull String f = "not null";

  void init() {
    // Executes before f is initialized
    System.out.println( f.length() ); //null deref!
  }
}
```

Listing 3.28: Java fields may be accessed before their initializer is run.

```
class A {
  A() {
    init();
  }

  @RawThis void init() { }
}

class B extends A {
  @NonNull String f = "not null";

  @RawThis void init() {
    if (f != null)
      System.out.println( f.length() );
  }
}
```

Listing 3.29: Using raw types to guard against null dereferences during object construction.

`@NonNull` fields have been assigned a non-null value by the end of construction. This check requires dataflow information, since it must reason about all paths through the constructors. This definite assignment analysis was easily added to the checker by defining a FlowFacts class using our dataflow framework.

The entire non-null checker, including raw types, required 136 lines of JAVACOP code consisting of 7 predicates and 12 rules. The two flow analyses used by the JAVACOP rules were built using our dataflow framework in a total of 147 lines of Java code.

3.7.1.2 Experience Using the Non-Null Checker

In Table 3.2 we present results of applying our non-null checker to two existing Java programs to make them safe from null dereferences. The first column contains results pertaining to an undergraduate project by one of the authors that uses Dijkstra's algorithm for determining shortest path on a given street map. The second column contains results from applying JAVACOP to itself, namely the pass that we added to the OpenJDK compiler for JAVACOP's rule enforcement. The table lists the size of each application and the number of object dereferences that the non-null type system must prove safe.

The table also lists the number of annotations required, both within the application itself and within depended-upon code: the Java standard library and (for JAVACOP) the rest of the `javac` compiler implementation. The number of annotations could be significantly reduced through the use of appropriate defaults [CJ07]. Such a non-null-by-default type system could be adapted from the non-null type system discussed here by changing the predicate `requiresNonNull`, which defines the non-null type, to require the lack of a `@Nullable` annotation instead of the presence of a `@NonNull` annotation.

Table 3.2: `@NonNull` annotation results for Dijkstra's algorithm implementation and JAVACOP. Additional code dependencies listed here include wrapper methods for library calls as well as additional annotated code. Nullity checks inserted due to lack of Java support for annotations on enhanced for loops and generics are listed under Java limitations.

	Dijkstra	*JavaCOP*
LOC	629	948
(add'l code dependency)		∼1000
Derefs	206	628
`@NonNull` annotations	83	92
(add'l code dependency)	43	100
`@RawThis` annotations	1	0
(add'l code dependency)	1	1
Nullity checks	46	93
bugs	7	7
false positives	0	22
unknown	0	14
Java limitations	39	50

(a)

```
public class JCList<T extends Object> {
          ⋮
  public boolean nonEmpty(){
    return tail != null;
  }
}
```

(b)

```
@NonNull JCList<String> list = ...;
for ( ; list.nonEmpty(); list = list.tail) {
  /* loop body */
}
```

(c)

```
@NonNull JCList<String> list = ...;
for ( ; list.nonEmpty(); ) {
  /* loop body */
  JCList<String> tail = list.tail;
  if (tail == null)
    throw new RuntimeException(...);
  list = tail;
}
```

Figure 3.5: Example code (b) demonstrating a false positive indicated by our `@NonNull` type system and how the code had to be modified (c) to appease the type system. This is a false positive because the type system is unaware of the invariant which the method `nonEmpty()` (a) ensures.

The "Nullity checks" category indicates the number of places in which we had to add an explicit test for non-nullness in order to typecheck successfully. These checks have been partitioned into several categories. The seven bugs in the Dijkstra application all have to do with improper handling of input files. If the files are not in the correct format, the implementation generates null pointers within its data structures, which can later be dereferenced. Seven bugs were also found in the JAVACOP source. For example, the expression `filename.getParentFile().toURL()` contains an error, since the method `getParentFile()` may return `null` for a malformed file descriptor. We list 14 nullity checks as "unknown". These checks all pertain to references to `javac Scope` objects. Our inspection of the `javac` code leads us to believe that the `Scope` objects are phased: initialized to `null` but, at some time before the JAVACOP pass, set to a non-null value. However, we found no conclusive evidence to support this belief and so left the checks uncategorized.

The code in Figure 3.5(b) illustrates an example false positive. The type system complains that the potentially null field `tail` is being assigned to the `@NonNull` variable list. However, the loop guard ensures that `tail` is non-null, as shown in Figure 3.5(a). We satisfy the type system by inserting a nullity check, as shown in Figure 3.5(c). This code also illustrates the need to introduce local variables, since the type system only supports flow sensitivity for local variables. Flow sensitivity for fields is more challenging due to the potential for aliasing and the possibility of concurrent access by multiple threads.

Finally, "Java limitations" lists nullity checks due to limitations in Java's annotation syntax. Most notably, type parameters cannot have annotations, so for example it is not possible to have a `List` of `@NonNull` strings. Therefore each time we access and use an element from such a list, a spurious nullity check is required. A current proposal would resolve this and related limitations in Java 7.0 [Ern07].

Testing the Checker After updating the non-null type system to support flow sensitivity, but before extending it to properly handle object initialization via raw types, I created a runtime checker for the non-null type system. I then created a test suite consisting of 79 unit tests and verified via the test harness that the pluggable type system and runtime checker agree on the results of all tests. The test harness was then further used during development of an extension to the type system to handle raw types.

First, a test case similar to the one in Listing 3.28 was created. It passed the static checker but failed the runtime checker, thus illustrating the unsoundness mentioned earlier. This test case was used during development to ensure that the resulting raw types checker indeed plugged the type hole. This process was repeated when adding flow sensitivity to check for definite assignment of `@NonNull` fields. Upon removing the JAVACOP rule requiring `@NonNull` fields to have initializers, the type system became unsound until the flow-sensitive checks were in place. The testing framework made it easy to concretely understand the type holes being fixed and to gauge progress toward these goals. In total, an additional 10 test cases were created when developing and testing the raw types extension to the first non-null checker.

3.8 Compiler Performance

To demonstrate that JAVACOP is a practical implementation, I measured its performance compiling a range of sample programs using a few rule sets. The sample programs include several well-known open source examples, as well as a simple Hello World example and the Java code generated to check the rules for the JAVACOP non-null type systems implementation. I compiled each program first with no rule sets, then with a confined type system, unique reference type system, and a non-null type system individually, and then finally with JAVACOP checking all three of these rule sets simultaneously.

The measurements were taken on a Dell Optiplex GX270, with an Intel Pentium IV 2.8GHz and 1.5GB RAM, running Fedora 8 in a KDE Konsole terminal. JAVACOP was run using the Sun Java HotSpot(TM) Client VM (build 1.6.0_05-b13, mixed mode, sharing). Each test was run five times and timed using the `time` command from the `bash` shell. The wall clock ('real') time was measured, the highest and lowest values were discarded, and the remaining three averaged to produce the final figure. Because the confined type system rules rely on annotations not present in most of the examples, I modified the rules so that *every* class would be checked as if it were confined. All numbers include the time required to print warnings, errors, and `ant` output to the screen.

Fig. 3.6 presents the results. For each configuration the upper row is time in seconds, while the lower row is the percentage slowdown over the baseline performance caused by using the given type system(s)—in each case lower is better. The key point from this table is that even when using multiple complex type systems, performing JAVACOP rule checking in addition to standard Java typechecking takes less than 1.7 times as long as Java typechecking alone. Closer inspection shows that most of the time is spent in the two type systems which use flow-sensitivity: the non-null and unique type systems. This is not surprising as those rules must make multiple visits over the AST and create a number of objects representing dataflow facts. The simpler confined type system only imposes overhead of between one and fifteen percent. I have not performed any optimizations on JAVACOP, nor investigated any kind of incremental compilation support, in order to obtain these numbers. Nonetheless, JAVACOP's performance demonstrates the practicality of its design.

System	Classes	No rule set	Confined	Unique	Nonnull	All
Hello World	1	0.6557	.7487	.7343	.7107	.7647
			14.2%	**8.4%**	**12.0%**	**16.6%**
Non-null rule set	4	1.2930	1.4603	1.4493	1.5610	1.8147
			12.9%	**12.1%**	**20.7%**	**40.3%**
Polyglot5	161	6.1060	7.0577	8.1930	7.4580	9.3163
			15.6%	**34.2%**	**22.1%**	**52.6%**
PMD	786	18.1073	19.0110	20.2160	20.6807	22.9273
4.2.4			**5.0%**	**11.6%**	**14.2%**	**26.6%**
JEdit	1085	10.8970	11.4950	14.6067	13.5540	18.1417
4.3 pre 15			**5.5%**	**34.1%**	**24.5%**	**66.6%**
Jython	1191	19.2883	20.8210	24.4997	25.4143	31.6790
2.5			**7.9%**	**27.0%**	**31.8%**	**64.2%**
OpenJDK	1281	11.0743	11.9663	14.9040	13.6270	17.4420
7-ea-src-b36			**8.1%**	**34.6%**	**23.1%**	**57.5%**

Figure 3.6: JAVACOP Compilation Times

3.9 Summary

In this chapter, I presented the JAVACOP framework for pluggable types in Java. The JAVACOP language allows developers to declaratively specify numerous non-trivial programming disciplines for both domain-specific disciplines and general-purpose type systems from the research literature. JAVACOP additionally provides a simple dataflow analysis API to let programmers utilize flow-sensitivity in their type systems when the JAVACOP language itself is not sufficient. The JAVACOP compiler enables these programming disciplines to be automatically enforced on Java programs. Via the Polyglot, SCJ, and non-null checker case studies, I showed how JAVACOP discipline checking can find bugs in real code. To ensure that JAVACOP programs also guarantee desirable program properties, I created the type system testing framework. This practical approach allows developers to specify desirable properties as runtime invariants which are used to instrumenting code and find programs from a test suite which successfully type check but still violate the invariants.

CHAPTER 4

CLARITY: Semantic Type Qualifiers for C

4.1 Introduction

While the JAVACOP framework provides a very expressive framework for defining disciplines, it does so to the detriment of automatic reasoning about the validity of a discipline. Using a test framework for validating a discipline against an intended invariant is a practical solution, but it does not guarantee complete coverage of the program space. Satisfying the expected and intended results of a discipline for a test suite is only a hint, not a proof, that the discipline is correct. In order to automate the verification process, the discipline language must be restricted so that theorem provers and constraint solvers can reason more directly about the rules of a discipline checker.

In this chapter, I present the CLARITY framework for semantic type qualifiers for the C programming language. The intent behind CLARITY's development was to show how a framework can be built which addresses automated invariant checking of disciplines. It offers developers an automated way of enforcing and validating disciplines via *type qualifiers*. Type qualifiers, in their simplest form, are lightweight type system refinements denoted in syntax via an annotation on a type. These qualified types are treated as subtypes of their respective unqualified types.

CLARITY supports *semantic* type qualifiers by providing a restrictive DSL focused on pattern matching over C expressions and statements. The intended semantic invariant of a discipline is supplied by the type system developer as a part of the discipline

specification using a stylized predicate syntax with built-in reasoning about program state information. The simplicity of the language and predicates allows qualifiers to be type checked, or even inferred; programs to be automatically instrumented for safe casting; and, sometimes, rules themselves to be inferred from the invariant predicate.

The type qualifier specifications in CLARITY enforce disciplines that restrict the values of C expressions. The type systems that can be defined in CLARITY are, thus, a strict subset of those that can be defined in JAVACOP. For instance, CLARITY can check simple versions of the non-null and untainted disciplines, but not the SCJ or Polyglot disciplines defined in Chapter 3. However, the CLARITY framework automatically enforces a subtyping strategy that allows rules to be more succinct. That is, at every point in a C program where a subtyping relationship is expected between two expressions–such as assignments, return statements, arguments to functions, etc.–CLARITY requires that the qualifiers on the subtype be a superset of the qualifiers on the supertype. This is the same behavior that had to be hand-coded in JAVACOP in Listing 3.4, but that is built-in to CLARITY. The listing below shows how a simple form of the *untainted* discipline would be specified in CLARITY.

```
value qualifier untainted(T Expr E)
```

The specification defines a new qualifier, `untainted`. As there are no other rules provided in the definition, only expressions explicitly annotated with the `untainted` qualifier will be considered untainted. As a result, CLARITY will automatically enforce that only expressions that have an `untainted` qualified type can be used when an `untainted` qualified type is expected.

CLARITY provides a novel combination of features to support the development of these semantic type qualifiers:

- *Declarative rule language.* CLARITY employs a declarative, pattern matching-based language for defining semantic type qualifiers. CLARITY's language was

created to be a simple, restricted language for defining new C type refinements that is easy for qualifier designers and programmers to understand and to define correctly and enables automated verification of qualifier rules against a semantic invariant. The language has a natural correspondence to the normal specification of syntax-directed typing rules and closely resembles pattern matching schemes found in languages such as ML.

- *Seamless integration with C.* CLARITY naturally allows type qualifiers to interact with C's existing type system. To enable efficient inference of qualifiers, it does whole program checking, as opposed to the modular style shown in JAVACOP. The typechecker is implemented as an module extension to the CIL infrastructure for C program analysis and transformation from Berkeley [NMR02]. This feature allows the CLARITY to be used in place of the standard C compiler in a development toolchain.

- *Support for inference.* Programmers often find it burdensome to fully annotate a program with necessary type qualifiers. To ease this burden, CLARITY provides whole program type inference for all expressions in a C program. This inference identifies all places where type qualifiers are necessary and can be used to statically identify invariants about program entities. CLARITY also supports automatic inference of type qualifier rules, which can enable a developer to make a more precise type system.

- *Automated qualifier validation.* To address discipline verification, CLARITY provides a novel *soundness* checker that uses an off-the-shelf automatic theorem prover to prove that qualifier rules ensure an associated semantic invariant. The soundness results are known to be valid given certain assumptions that are made about the C runtime states.

While not as diverse a set of disciplines as with JAVACOP, I have defined and verified a number of non-trivial type qualifiers in CLARITY and used them to detect real errors and infer interesting invariants about open-source C programs. CLARITY performs reasonably well when inferring multiple type qualifiers, although it shows some expected scaling issues with larger programs.

The rest of this chapter is organized as follows. Section 4.2 introduces the design of CLARITY's rule language through a number of examples and introduces the concepts of *value* and *flow* type qualifiers. Section 4.3 explains the basic type checking scheme of CLARITY without inference. Section 4.4 discusses the extension to the basic type checking scheme necessary to support inference of *value* qualifiers. Section 4.5 describes the automated validation scheme of type qualifier rules,and Section 4.6 describes the experience using CLARITY qualifier checking and inference on open-source C projects.

4.2 Semantic Type Qualifiers

The CLARITY framework supports the definition of a common class of qualifiers known as *value* qualifiers. Value qualifiers, such as `pos` and `nonnull`, pertain only to the value of an expression.

4.2.1 Value Qualifiers

Listing 4.1 illustrates a definition of the value qualifier `pos` in the CLARITY framework, which can be used to statically track positive integers. Line 1 of the listing declares `pos` to be a new value qualifier applicable to expressions of type `int`. It also declares a variable `E`, which is used in the rest of the qualifier's definition. Each variable declaration includes a type and a *classifier*. The declared classifier `Expr` for `E`

```
1 value qualifier pos(int Expr E)
2   case E of
3       decl int Const C:
4         C, where C > 0
5     | decl int Expr E1, E2:
6         E1 * E2, where pos(E1) && pos(E2)
7     | decl int Expr E1:
8         -E1, where neg(E1)
9   invariant value(E) > 0
```

Listing 4.1: A user-defined type qualifier specification (i.e., declaration and associated type rules) for positive integers.

```
int pos gcd(int pos n, int pos m);
int pos lcm(int pos a, int pos b) {
  int pos d = gcd(a, b);
  int pos prod = a * b;
  return (int pos) (prod / d);
}
```

Listing 4.2: Example code using the `pos` type qualifier.

indicates that during typechecking of a C program, `E` will be instantiated with side-effect-free program expressions. The declared type for `E` constrains such expressions to have type `int`. In addition to the classifier `Expr`, the CLARITY framework supports the classifiers `Const`, `LValue`, and `Var`, which represent C constants, l-values, and variables, respectively. As CLARITY is implemented as an extension to the CIL infrastructure for C program analysis and transformation [NMR02], it performs qualifier checking over programs in CIL's intermediate language, which cleanly distinguishes expressions, which are side-effect-free, from instructions.

Given the declaration on line 1 of Listing 4.1, programmers may now annotate their programs with the `pos` qualifier, as shown in the C code in Listing 4.2. The `lcm` procedure in the figure computes the least-common multiple of two integers. The `pos` qualifier is used to specify that the two arguments should be positive integers and to ensure that the return value is also positive. To handle nested qualifiers unambiguously, we use a postfix notation, whereby a qualifier qualifies the entire type to its left. A type may be annotated with multiple user-defined qualifiers; their order is irrelevant.

4.2.1.1 Type Rules

Line 1 of Listing 4.1 declares the new `pos` qualifier, but it does not indicate how this qualifier should be used during typechecking. This is the role of the `case` block beginning on line 2, which uses a form of pattern matching to indicate a subset of expressions that can be given the type `int pos`. For example, the clause in lines 3-4 indicates that a positive integer constant may be given the type `int pos`. The clause first declares the variable `C`, which ranges over integer constants from the underlying program, for use in the rest of the clause. It then specifies the pattern `C`, to indicate the syntactic form of the expression. Finally, the predicate `C > 0` further constrains an expression that matches the pattern.

```
value qualifier neg(int Expr E)
  case E of
      decl int Const C:
        C, where C < 0 \\
    | decl int Expr E1, E2:
        E1 * E2, where neg(E1) && pos(E2)
    | decl int Expr E1, E2:
        E1 * E2, where pos(E1) && neg(E2)
    | decl int Expr E1:
        -E1, where pos(E1)
  invariant value(E) < 0
```

Listing 4.3: A user-defined type qualifier specification for negative integers. This specification demonstrates a mutually recursive relationship with the `pos` qualifier defined in Listing 4.1.

Type rules like the first `case` clause of Listing 4.1 can be simulated in simpler type qualifier systems like CQUAL [FFA99] by annotating all positive integers in a program with a `pos` assumption. However, the `case` clauses in CLARITY are more general. For example, the clause on lines 5-6 specifies that an expression that is a product of two expressions of type `int pos` can also be given the type `int pos`. This kind of recursive type rule would be quite difficult to manually encode using `pos` assumptions. The final `case` clause illustrates that the definition of a qualifier can depend on other qualifiers. That clause specifies that a negation expression can be given type `int pos` if the negated expression can be given type `int neg`, where `neg` is another user-defined qualifier. In fact, qualifier definitions can be mutually recursive. For example, the definition of `neg` has rules that refer to `pos`, as shown in Listing 4.3.

The syntax for expression patterns in `case` clauses is defined by the following grammar:

$$P \quad ::= \quad X \mid *X \mid \&X \mid \texttt{new} \mid \mathit{uop}\ P \mid P\ \mathit{bop}\ P \mid \texttt{return}\ P$$

Here X ranges over *variable patterns*, which have a declared type and classifier (e.g.,

`int Expr`) restricting the kinds of program fragments that may match. The pattern `new` matches against calls to memory allocation routines like `malloc`Various unary and binary operations may also be matched against. Finally, the *return pattern* allows users to specify special-purpose type rules for the expression returned from a procedure. The predicate after (the optional) `where` in a `case` clause may include operations on constants and on variable patterns with classifier `Const`, qualifier checks on expressions and patterns, and conjunctions and disjunctions of these kinds of predicates.

Each clause of a `case` block can be viewed as an *introduction* type rule for a qualified type, since the clause specifies conditions under which an expression may be assigned that qualified type. For example, the second clause in Listing 4.1 is akin to the following type system rule:

$$\frac{\Gamma \vdash E_1 : \texttt{int pos} \qquad \Gamma \vdash E_2 : \texttt{int pos}}{\Gamma \vdash E_1 * E_2 : \texttt{int pos}}$$

The semantics of the `case` clauses are formalized in a companion tech report [CMM04].

The CLARITY typechecker uses the type rules defined by `case` blocks, along with a set of standard rules for typechecking constructs like variable references, procedure calls, and assignments, to perform qualifier checking. Such checking validates the qualifier annotations supplied by the programmer, which represent the programmer's assumptions about when particular invariants hold. For example, if a variable's type is annotated with qualifier *q*, then qualifier checking only succeeds if every expression used as the right-hand side in an assignment to that variable can be determined to have that same type.

Consider again the `lcm` procedure in figure 4.2. As usual, typechecking an assignment statement involves obtaining the types of each side and checking that they match. The assignment to `d` typechecks successfully because both sides of the assignment have type `int pos`: the right-hand side is shown to have this type by the standard

```
value qualifier nonzero(int Expr E)
  case E of
      decl int Const C:
        C, where C != 0
    | decl int Expr E1:
        E1, where pos(E1)
    | decl int Expr E1:
        E1, where neg(E1)
    | decl int Expr E1, E2:
        E1 * E2, where nonzero(E1) && nonzero(E2)
  restrict
      decl int Expr E1, E2:
        E1 / E2, where nonzero(E2)
  invariant value(E) != 0
```

Listing 4.4: A CLARITY type qualifier specification for nonzero integers.

type rule for procedure calls, given the declared type signature of `gcd`. The assignment to `prod` also typechecks successfully, because the `case` clause on lines 5-6 allows `a * b` to be given the type `int pos`. Because of their declared types, we statically know that both `prod` and `d` are positive, but this information is not sufficient to show that the expression `(prod / d)` is also positive. Indeed, the type rules for `pos` are not able to derive the type `int pos` for that expression. Therefore, the programmer must insert a cast to satisfy the typechecker, because of the declared return type of `lcm`.

A `case` block specifies when an expression may be given a qualified type. Specification developers may also want enhance the precision of existing typing rules from the base C type system using type qualifier expressions. CLARITY provides a `restrict` block for this purpose, an example of which is shown in the definition for a `nonzero` qualifier in Listing 4.4. The syntax of a `restrict` clause is identical to that of a `case` clause. A `restrict` clause specifies that any expression in a given program that matches the clause's pattern must also satisfy the clause's predicate. The `restrict` clause for `nonzero` overrides the base rule for typechecking division expressions to

require that the denominator have the type `int nonzero` (rather than simply `int`). In this way, potential divide-by-zero errors can be detected statically instead of being found at runtime.

For example, consider again the `lcm` procedure in Listing 4.2. If the extensible typechecker is given the definition of `nonzero` in addition to that of `pos`, it will use `nonzero`'s `restrict` clause to check the division in the last statement of `lcm`'s body. The `restrict` clause requires that `d` have the type `int nonzero`. By the second `case` clause for `nonzero` in Listing 4.4, any expression of type `int pos` also has the type `int nonzero`. Since `d` is declared to have the type `int pos`, that `case` clause allows the `restrict` check to succeed.

The `restrict` clause is analogous to qualifier assertions in CQUAL [FFA99]. For example, the `restrict` type rule in Listing 4.4 could be simulated by annotating the denominator in each division in a program with a `nonzero` assertion. However, the `restrict` clause is more general. For example, the predicate in a `restrict` clause may contain conjunctions and disjunctions of qualifier checks.

4.2.1.2 Subtyping

It is natural to consider `int pos` to be a subtype of `int`. For example, this subtyping relationship would allow the following code snippet to typecheck:

```
int pos x = 3;
int y = x;
```

The CLARITY typechecker considers *all* value-qualified types to be subtypes of their associated unqualified types. More precisely, if q is a value qualifier and τ is a (possibly qualified) type, then $\tau\ q$ is considered to be a subtype of τ.

The rest of the supported subtyping rules are standard. As usual, care must be taken in the presence of pointers [FFA99]. For example, it would be unsound to consider `int`

`pos*` to be a subtype of `int*`, because that would allow the following code, which stores a negative number in a variable of type `int pos`, to typecheck:

```
int pos x = 3;
int* p = &x;
*p = -1;
```

A companion technical report contains the formal definition of the subtyping relation in CLARITY [CMM04].

CLARITY does not support explicit subtype declarations between two user-defined qualifiers. However, such subtype relationships can be encoded using the `case` block. For example, the second clause in the `case` block of `nonzero`'s definition in Listing 4.4 effectively declares `pos` to be a subtype of `nonzero`: any expression of type `int pos` may also be considered to have type `int nonzero`.

4.2.1.3 Semantic Invariants

Line 9 of Listing 4.1 uses an `invariant` clause to define the `pos` qualifier's associated run-time invariant. The invariant is a predicate that is implicitly defined in the context of an arbitrary run-time execution state. Let us denote this execution state by ρ. The `value` predicate is provided by our framework and represents the value of a given expression in ρ. Therefore, the invariant for `pos` indicates that the value of an expression of type `int pos` should be greater than zero, in any run-time execution state.

The limitations of the CLARITY language for writing type rules, and of static typechecking in general, sometimes requires programmers to insert casts in order for qualifier checking to succeed. To retain soundness with respect to a specified invariant in these cases, CLARITY's typechecker instruments programs with a run-time check for each cast to a value-qualified type. Each run-time check tests whether the expression being cast satisfies the cast-to qualifier's invariant. In the implementation, a fatal error

is signaled if the test fails. For example, consider the cast in the last statement of `lcm` in Listing 4.2. At run time, a check ensures that the value of `(prod / d)` is in fact greater than zero.

4.2.1.4 Flow Qualifiers

Some common kinds of qualifiers are used solely to restrict the flow of values in a program. For example, the *untainted* discipline used as a running example throughout this dissertation essentially tracks the flow of data coming from trustworthy and potentially untrustworthy sources. For soundness, the only requirement is that `tainted` data never flows where `untainted` data is expected. The `untainted` qualifier can help to statically detect SQL injection attacks, as discussed in Chapter 1, but it is equally useful for tracking format-string vulnerabilities in calls to C's `printf` and related procedures [STF01]. Another example of flow qualifiers include the qualifiers `user` and `kernel` which can be used to statically ensure that user pointers are never dereferenced in kernel space [JW04].

The listing below shows a CLARITY specification for a simple untainted analysis in the CLARITY framework.

```
value qualifier untainted (T Expr E)
```

As the specification is defined, the `untainted` qualifier can qualify any type `T`. Flow qualifiers like `untainted` are a degenerate form of value qualifier in CLARITY: since the `untainted` qualifier has no `case` block, the only way to introduce an expression of type `T untainted` is with a cast, to explicitly mark the expression as being trustworthy. The qualifier also lacks an explicit run-time invariant as the invariant language is not expressive enough to capture runtime properties associated with the discipline. Proper value flow is guaranteed "for free" because `T untainted` is a subtype of `T` but not vice versa. Therefore, untainted data can flow where arbitrary data is expected, but

not vice versa.

As discussed earlier, the `untainted` qualifier can help find errors in calls to various SQL APIs for C. For example, the DB2 database management system from IBM provides a function `SQLExecDirect()` for directly executing a SQL statement on a database. The second argument to `SQLExecDirect` is the SQL statement that will be executed on the database. C itself does not verify, either at compile time or run time, that calling `SQLExecDirect` with an arbitrary SQL statement will not have unintended effects on the database. To ensure that only well-behaved statements are executed on the database, a programmer can use a type signature for `SQLExecDirect` that requires the second formal parameter to have type `char* untainted`.

For example, suppose `unsafe_stmt` is a SQL statement which has been completed by getting input from an arbitrary user via the `getName` function. Also, suppose that `safe_stmt` uses the same SQL statement fragment but fills it with a known safe string literal. Then typechecking with the simple untainted type system defined above on the following code allows the first safe call to `SQLExecDirect`, but signals an error for the unsafe second call:

```
char unsafe_stmt[MAX_SQL_SIZE];
sprintf(unsafe_stmt,
   "SELECT * FROM users WHERE name='%s';",getName());
char* untainted safe_stmt =
   (char* untainted) "SELECT * FROM users WHERE name='shane';";

result = SQLExecDirect(hstmt, safe_stmt, ...); // OK
result2 = SQLExecDirect(hstmt, unsafe_stmt, ...); // ERROR
```

It may be useful to explicitly annotate some expressions as being possibly `tainted` for code readability. For example, it might be helpful to declare the header for the `main` function of a program as follows, to explicitly indicate that command-line arguments should not be trusted:

```
int main(int argc, char* tainted argv[]);
```

The definition of `tainted` in the listing below allows the `tainted` qualifier to be given to any expression.

```
value qualifier tainted(T Expr E)
  case E of E
```

The lone `case` clause allows any expression to be considered `tainted`, effectively making `T tainted` a supertype of `T` (and hence also of `T untainted`). Because of the implicit subtyping relation for value qualifiers, it is also the case that `T tainted` is a subtype of `T`, so those types are essentially equivalent.

Although the versions of `tainted` and `untainted` shown above are degenerate, they could easily be augmented. For example, a user could decide that all constants should be trusted, adding a `case` clause to the definition of `untainted` as follows:

```
  case E of decl T Const C: C
```

This rule would, for example, obviate the need for the cast in the assignment to `safe_stmt` in the previous C code snippet using `SQLExecDirect`. This is also equivalent to one of the predicate declarations from the JAVACOP rules for untainted shown in Listing 3.9.

4.3 Extensible Typechecking

The CLARITY typechecker takes as arguments a C program and a set of qualifier specifications in the language described in the previous section. The typechecker then performs qualifier checking on the program as directed by the qualifier specifications' type rules. The extensible typechecker also uses value qualifiers' declared invariants to instrument the program with run-time checks for casts involving value qualifiers, as mentioned in Section 4.2.1.3.

The CLARITY typechecker is implemented as a module in CIL [NMR02], a front

end for C written in OCaml [Rem98]. CIL parses C code into an abstract syntax tree (AST) format and provides a framework for performing passes over this AST. After qualifier checking, the AST is output as C code and the `gcc` compiler performs ordinary C typechecking and code generation.

4.3.1 Annotating Programs

CLARITY makes use of `gcc` *attributes*, which are tags that can be associated with types (and other program entities) similar, but more flexible than, Java metadata annotations as seen in Chapter 3, to introduce qualifiers into C programs. CIL supports `gcc` attributes and maintains them in the generated AST for a program. A type attribute follows the type name and has the following syntax:

`__attribute__((`*attribute name*`))`

Instead of directly using this rather unwieldy syntax, users can define C macros to replace the name of the qualifier with the full syntax. Such macros are used in the C language examples of Section 4.2. For example, the qualifier `pos` used in Listing 4.2 is defined as follows:

```
#define pos __attribute__((pos))
```

4.3.2 Qualifier Checking with CIL

To enforce a qualifier specification, the CLARITY typechecker traverses the provided CIL AST, applying user-defined type rules to applicable program fragments. Any type errors found during qualifier checking are provided to the programmer as warnings, but compilation is allowed to continue.

To implement qualifier checking, I created a set of OCaml datatypes to represent the expression patterns and predicates that are allowed in user-defined type rules. For

example, consider the `case` clause on lines 5-6 in Listing 4.1. The expression pattern is represented internally as follows:

```
Binop(Mult, Expr("E1"), Expr("E2"))
```

The clause's predicate is similarly represented as follows:

```
And(Qual("pos", Expr("E1")),
    Qual("pos", Expr("E2")))
```

Consider the application of this type rule to the right-hand side of the assignment to `prod` in Listing 4.2. First, the typechecker matches the expression pattern against the CIL AST for `a * b`. The match succeeds and produces bindings for variables in the pattern: `E1` is bound to the expression `a` and `E2` is bound to the expression `b`. Finally, the rule's predicate is evaluated, after replacing each pattern variable with the C program fragment to which it is bound. In this example, the predicate is satisfied if `a` and `b` can recursively satisfy the qualifier `pos`. The other kinds of type rules are represented and checked similarly.

4.3.3 Interacting with C

CLARITY allows types to be annotated with qualifiers wherever they appear. For example, the types of `struct` fields may be qualified, and the typechecker will check that they obey the user-defined type rules. Fields of unions may also be given qualified types, but the usual unsoundness for C unions makes qualifier checking in this case unsound as well.

As is often the case for C program analyses, CLARITY assumes a logical model of memory. In particular, the type of `p+i`, where `p` is a pointer and `i` is an integer, is assumed to be the same as the type of `p`. This assumption is unsound, but in practice it removes a large source of spurious type errors; for example arising from pointer arithmetic for array indexing.

Another source of spurious type errors arises from invoking procedures in the C standard library, since their argument and result types are not annotated with user-defined qualifiers. A CLARITY user can mitigate this problem by writing header files that contain alternate signatures for library procedures, which replace the procedures' ordinary signatures via `gcc` command-line macros. Macros from the standard library are also problematic. When these macros are expanded by the C pre-processor, CLARITY produces type errors because the macros' bodies are not properly annotated. Short of creating new external versions of these macros, there is little recourse in prevent such spurious errors. Typechecking can also be unsound because it allows variables to be used before being initialized (according to C conventions) and does not model arithmetic overflow.

4.4 Qualifier Inference

Up to this point I have discussed CLARITY's qualifier *checking* capabilities: that is, all variables have had to be explicitly annotated with their qualifiers in order to typecheck correctly. In this section, I show how CLARITY supports qualifier *inference* in the presence of user-defined qualifier rules. I formalize qualifier inference for a simply-typed lambda calculus with references and user-defined qualifiers, as defined by the following grammar:

$$
\begin{array}{rcl}
e & ::= & c \mid e_1 + e_2 \mid x \mid \lambda x : \tau . e \mid e_1\ e_2 \mid \texttt{ref}\ e \mid e_1 := e_2 \mid !e \mid \texttt{assert}(e, q) \\
\tau & ::= & \texttt{int} \mid \tau_1 \rightarrow \tau_2 \mid \texttt{ref}\ \tau
\end{array}
$$

Let Q be the set $\{q_1, \ldots, q_n\}$ of user-defined qualifiers in a program. Sets of qualifiers from Q form a natural lattice, with partial order $\supseteq$, least-upper-bound function $\cap$, and greatest-lower-bound function $\cup$. I denote elements of this lattice by metavariable

l; qualified types are ranged over by metavariable ρ and are defined as follows:

$$\rho \ ::= \ l\,\phi \qquad \phi ::= \texttt{int} \mid \rho_1 \rightarrow \rho_2 \mid \texttt{ref}\ \rho$$

I present both a type system and a constraint system for qualifier inference and describe an algorithm for solving the generated constraints. The type system defines what type qualifiers are necessary in order for a program to be well-typed, while the constraint system provides an efficient implementation strategy for determining those qualifiers. I assume that bound variables in expressions are annotated with unqualified types τ. It is possible to combine qualifier inference with type inference, but separating them simplifies the presentation.

4.4.1 Formal Qualifier Rules

I formalize the `case` rules as defining two kinds of relations. First, some `case` clauses have the effect of declaring a specificity relation between qualifiers. I formalize these rules as defining axioms for a relation of the form $q_1 \triangleright q_2$. For example, the second `case` clause in Listing 4.4 would be represented by the axiom $\texttt{pos} \triangleright \texttt{nonzero}$. I use $\triangleright^*$ to denote the reflexive, transitive closure of the user-defined $\triangleright$ relation, and we require $\triangleright^*$ to be a partial order.

The other kind of `case` clause uses a pattern to match on a constructor (e.g., +), and the clause determines the qualifier of the entire expression based on the qualifiers of the immediate subexpressions. I formalize these rules as defining relations of the form R_p^q, where q is a qualifier and p represents one of the constructors in the formal language, ranging over integer constants and the symbols +, λ, and `ref`. The arity of each relation R_p^q is the number of immediate subexpressions of the constructor represented by p, and the domain of each argument to the relation is Q. Each `case` clause is formalized through axioms for these relations. For example, the fourth `case` clause

in Listing 4.4 would be represented by the axiom $R_{*}^{\texttt{nonzero}}(\texttt{nonzero},\texttt{nonzero})$ (if the formal language contained the * operator). The first `case` clause in Listing 4.4 would be formalized through the (conceptually infinite) set of axioms $R_1^{\texttt{nonzero}}()$, $R_2^{\texttt{nonzero}}()$, etc. For simplicity of presentation, I assume that each sub-expression is required to satisfy only a single qualifier. In fact, the implementation allows each sub-expression to be constrained to satisfy a set of qualifiers, and it is straightforward to update the formalism to support this ability.

Finally, I formalize the `restrict` rules with an expression of the form $\texttt{assert}(e,q)$, which requires the type system to ensure that the top-level qualifier on expression e's type includes qualifier q. For example, the `restrict` rule in Listing 4.4 is modeled by replacing each denominator expression e in a program with $\texttt{assert}(e,\texttt{nonzero})$. The `assert` expression can also be used to model explicit qualifier annotations in programs.

4.4.2 The Type System

In this section I formalize qualifier inference as a type system over the simply-typed lambda calculus with references and user-defined qualifiers as defined previously. The previous section provided some insight into how CLARITY interacts with C via pattern matching, but this formalism defines how to infer the qualifier necessary for a well-typed program.

The qualifier type system is presented in Figure 4.2, and the set of axioms A representing the user-defined qualifier rules are implicitly considered to augment this formal system. As usual, metavariable Γ ranges over type environments, which map variables to qualified types. The rule for $\texttt{assert}(e,q)$ infers a qualified type for e and then checks that q is in the top-level qualifier of this type. The *strip* function used in the rule for lambdas removes all qualifiers from a qualified type ρ, producing an unquali-

$$\frac{l_1 \supseteq l_2}{l_1\,\texttt{int} \leq l_2\,\texttt{int}} \qquad \frac{l_1 \supseteq l_2 \qquad \rho \leq \rho' \qquad \rho' \leq \rho}{l_1\ \texttt{ref}\ \rho \leq l_2\ \texttt{ref}\ \rho'} \qquad \frac{l_1 \supseteq l_2 \qquad \rho_2 \leq \rho_1 \qquad \rho_1' \leq \rho_2'}{l_1(\rho_1 \to \rho_1') \leq l_2(\rho_2 \to \rho_2')}$$

Figure 4.1: Formal subtyping rules for qualified types.

$$\frac{l = \{q \mid R_c^{q'}() \wedge q' \rhd^* q\}}{\Gamma \vdash c : l\ \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : l_1\ \texttt{int} \qquad \Gamma \vdash e_2 : l_2\ \texttt{int} \qquad l = \{q \mid R_+^{q'}(q_1,q_2) \wedge q_1 \in l_1 \wedge q_2 \in l_2 \wedge q' \rhd^* q\}}{\Gamma \vdash e_1 + e_2 : l\ \texttt{int}}$$

$$\frac{\Gamma(x) = \rho}{\Gamma \vdash x : \rho} \qquad \frac{\mathit{strip}(\rho_1) = \tau_1 \qquad \Gamma, x : \rho_1 \vdash e : \rho_2 \qquad \rho_2 = l_2\ \phi_2 \qquad l = \{q \mid R_\lambda^{q'}(q_2) \wedge q_2 \in l_2 \wedge q' \rhd^* q\}}{\Gamma \vdash \lambda x : \tau_1.e : l(\rho_1 \to \rho_2)}$$

$$\frac{\Gamma \vdash e_1 : l(\rho_2 \to \rho) \qquad \Gamma \vdash e_2 : \rho_2}{\Gamma \vdash e_1\ e_2 : \rho} \qquad \frac{\Gamma \vdash e : \rho \qquad \rho = l_0\ \phi_0 \qquad l = \{q \mid R_{\texttt{ref}}^{q'}(q_0) \wedge q_0 \in l_0 \wedge q' \rhd^* q\}}{\Gamma \vdash \texttt{ref}\ e : l\ \texttt{ref}\ \rho}$$

$$\frac{\Gamma \vdash e_1 : l\ \texttt{ref}\ \rho \qquad \Gamma \vdash e_2 : \rho}{\Gamma \vdash e_1 := e_2 : \rho} \qquad \frac{\Gamma \vdash e : l\ \texttt{ref}\ \rho}{\Gamma \vdash !e : \rho} \qquad \frac{\Gamma \vdash e : \rho \qquad \rho = l\ \phi \qquad q \in l}{\Gamma \vdash \texttt{assert}(e,q) : \rho}$$

$$\frac{\Gamma \vdash e : \rho' \qquad \rho' \leq \rho}{\Gamma \vdash e : \rho}$$

Figure 4.2: Formal qualifier inference rules.

$$\begin{aligned}
\alpha_1 \texttt{int} \sqsubseteq \alpha_2 \texttt{int} &\equiv \{\alpha_1 \supseteq \alpha_2\} \\
\alpha_1 \texttt{ref}\ \delta_1 \sqsubseteq \alpha_2 \texttt{ref}\ \delta_2 &\equiv \{\alpha_1 \supseteq \alpha_2\} \cup \delta_1 \sqsubseteq \delta_2 \cup \delta_2 \sqsubseteq \delta_1 \\
\alpha_1(\delta_1 \to \delta_1') \sqsubseteq \alpha_2(\delta_2 \to \delta_2') &\equiv \{\alpha_1 \supseteq \alpha_2\} \cup \delta_2 \sqsubseteq \delta_1 \cup \delta_1' \sqsubseteq \delta_2'
\end{aligned}$$

Figure 4.3: Converting type constraints into set constraints for CLARITY.

fied type τ.

The main novelty in the type system is the consultation of the axioms in A to produce the top-level qualifiers for constructor expressions. For example, consider the first rule in Figure 4.2, which infers the qualifiers for an integer constant c using a set comprehension notation. The resulting set l includes all qualifiers q' such that the $R_c^{q'}()$ relation holds (according to the axioms in A), as well as all qualifiers q that are "less specific" than such a q' as defined by the $\triangleright^*$ relation. In this way, the rule finds all possible qualifiers that can be proved to hold given the user-defined `case` clauses. The subsumption rule at the end of the figure can then be used to forget some of these qualifiers, via the subtyping rules in Figure 4.1. The inference of top-level qualifiers is similar for the other constructors, except that consultation of the R relation makes use of the top-level qualifiers inferred for the immediate subexpressions.

4.4.3 The Constraint System

While the formal type system presented previously clearly defines how type qualifiers and their associated axioms can be integrated into the base type system, it is not an effective approach for implementing qualifier inference. In this section I describe a constraint-based algorithm for qualifier inference. The key novelty is the use of a specialized form of *conditional constraints* to represent the effects of user-defined qualifier rules. The metavariable α represents *qualifier variables*, and generated constraints have the following forms:

$$\frac{\alpha' \text{ fresh} \qquad \delta' = \alpha'\ \texttt{int} \qquad \delta = \mathit{refresh}(\delta')}{\kappa \vdash c : \delta \mid \delta' \sqsubseteq \delta \cup \{C_c^q(\alpha') \mid q \in Q\}}$$

$$\frac{\begin{array}{c}\kappa \vdash e_1 : \alpha_1\ \texttt{int} \mid C_1 \qquad \kappa \vdash e_2 : \alpha_2\ \texttt{int} \mid C_2 \\ \alpha' \text{ fresh} \qquad \delta' = \alpha'\ \texttt{int} \qquad \delta = \mathit{refresh}(\delta')\end{array}}{\kappa \vdash e_1 + e_2 : \delta \mid C_1 \cup C_2 \cup \delta' \sqsubseteq \delta \cup \{C_+^q(\alpha_1, \alpha_2, \alpha') \mid q \in Q\}}$$

$$\frac{\begin{array}{c}\kappa, x : \delta_1 \vdash e : \delta_2 \mid C \qquad \delta_1 = \mathit{embed}(\tau_1) \qquad \delta_2 = \alpha_2\ \varphi_2 \\ \alpha' \text{ fresh} \qquad \delta' = \alpha'(\delta_1 \to \delta_2) \qquad \delta = \mathit{refresh}(\delta')\end{array}}{\kappa \vdash \lambda x : \tau_1 . e : \delta \mid C \cup \delta' \sqsubseteq \delta \cup \{C_\lambda^q(\alpha_2, \alpha') \mid q \in Q\}}$$

$$\frac{\kappa(x) = \delta' \qquad \delta = \mathit{refresh}(\delta')}{\kappa \vdash x : \delta \mid \delta' \sqsubseteq \delta} \qquad \frac{\begin{array}{c}\kappa \vdash e_1 : \alpha(\delta_2 \to \delta') \mid C_1 \qquad \kappa \vdash e_2 : \delta_2' \mid C_2 \\ \delta = \mathit{refresh}(\delta')\end{array}}{\kappa \vdash e_1\ e_2 : \delta \mid C_1 \cup C_2 \cup \delta_2' \sqsubseteq \delta_2 \cup \delta' \sqsubseteq \delta}$$

$$\frac{\begin{array}{c}\kappa \vdash e : \delta_0 \mid C \qquad \delta_0 = \alpha_0\ \varphi_0 \\ \alpha' \text{ fresh} \qquad \delta' = \alpha'\ \texttt{ref}\ \delta_0 \qquad \delta = \mathit{refresh}(\delta')\end{array}}{\kappa \vdash \texttt{ref}\ e : \delta \mid C \cup \delta' \sqsubseteq \delta \cup \{C_{\texttt{ref}}^q(\alpha_0, \alpha') \mid q \in Q\}}$$

$$\frac{\begin{array}{c}\kappa \vdash e_1 : \alpha\ \texttt{ref}\ \delta' \mid C_1 \qquad \kappa \vdash e_2 : \delta'' \mid C_2 \\ \delta = \mathit{refresh}(\delta')\end{array}}{\kappa \vdash e_1 := e_2 : \delta \mid C_1 \cup C_2 \cup \delta'' \sqsubseteq \delta' \cup \delta' \sqsubseteq \delta} \qquad \frac{\begin{array}{c}\kappa \vdash e : \alpha\ \texttt{ref}\ \delta' \mid C \\ \delta = \mathit{refresh}(\delta')\end{array}}{\kappa \vdash !e : \delta \mid C \cup \delta' \sqsubseteq \delta}$$

$$\frac{\begin{array}{c}\kappa \vdash e : \delta' \mid C \qquad \delta' = \alpha\ \phi \\ \delta = \mathit{refresh}(\delta')\end{array}}{\kappa \vdash \texttt{assert}(e, q) : \delta \mid C \cup \{q \in \alpha\} \cup \delta' \sqsubseteq \delta}$$

Figure 4.4: Formal constraint generation rules for qualifier inference in CLARITY.

$$\alpha \supseteq \alpha \qquad q \in \alpha \qquad q \in \alpha \Rightarrow \bigvee(\bigwedge q \in \alpha)$$

Given a set C of constraints, let S be a mapping from the qualifier variables in C to sets of qualifiers. S is said to be a *solution* to C if S satisfies all constraints in C. S is said to be the *least solution* to C if for all solutions S' and qualifier variables α in the domain of S and S', $S(\alpha) \supseteq S'(\alpha)$. It is easy to show that if a set of constraints C in the above form has a solution, then it has a unique least solution.

4.4.3.1 Constraint Generation

I formalize constraint generation by a judgment of the form $\kappa \vdash e : \delta \mid C$. Here C is a set of constraints in the above form, and the metavariable δ represents qualified types whose qualifiers are all qualifier variables:

$$\delta \ ::= \ \alpha\,\varphi \qquad \varphi ::= \texttt{int} \mid \delta_1 \to \delta_2 \mid \texttt{ref}\ \delta$$

The metavariable κ denotes type environments that map program variables to qualified types of the form δ.

The inference rules defining this judgment are shown in Figure 4.4. The *embed* function adds fresh qualifier variables to an unqualified type τ in order to turn it into a qualified type δ, and $\mathit{refresh}(\delta)$ is defined as $\mathit{embed}(\mathit{strip}(\delta))$. To keep the constraint generation purely syntax-directed, subsumption is "built in" to each rule: the *refresh* function is used to create a fresh qualified type δ, which is constrained by a subtype constraint of the form $\delta' \sqsubseteq \delta$. Subtype constraints are also generated for applications and assignments, as usual. Subtype constraints are treated as a shorthand for a set of qualifier-variable constraints, as shown in Figure 4.3.

Each rule for an expression with top-level constructor p produces one conditional constraint per qualifier q in Q, denoted C_p^q. Informally, the constraint C_p^q *inverts* the user-defined qualifier rules, indicating all the possible ways to prove that an expression

with constructor p can be given qualifier q according to the axioms in A. For example, both the second and third `case` clauses in Listing 4.4 can be used to prove that a product `a*b` has the qualifier `nonzero`, so our implementation of constraint generation in CLARITY produces the following conditional constraint:

$$\texttt{nonzero} \in \alpha_{\texttt{a*b}} \Rightarrow ((\texttt{nonzero} \in \alpha_{\texttt{a}} \wedge \texttt{nonzero} \in \alpha_{\texttt{b}}) \vee (\texttt{pos} \in \alpha_{\texttt{a*b}}))$$

More formally, let $zip(R_p^q(q_1,\ldots,q_m),\alpha_1,\ldots,\alpha_m)$ denote the constraint $q_1 \in \alpha_1 \wedge \ldots \wedge q_m \in \alpha_m$. Let $\{a_1,\ldots,a_u\}$ be all the axioms in A for the relation R_p^q, and let $\{q_1,\ldots,q_v\} = \{q' \in Q \mid q' \triangleright q\}$. Then $C_p^q(\alpha_1,\ldots,\alpha_m,\alpha')$ is the following conditional constraint:

$$q \in \alpha' \Rightarrow (\bigvee_{1\le i\le u} zip(a_i,\alpha_1,\ldots,\alpha_m) \vee \bigvee_{1\le i\le v} q_i \in \alpha')$$

This constraint system is actually equivalent to the type system presented in the previous subsection; details of the proof of the systems' equivalence are presented in a companion technical report [CMM05b].

Theorem: $\emptyset \vdash e : \rho$ if and only if $\emptyset \vdash e : \delta \mid C$ and there exists a solution S to C such that $S(\delta) = \rho$.

4.4.3.2 Constraint Solving

The qualifier inference constraints are solved by a graph-based propagation algorithm, which either determines that the constraints are unsatisfiable or produces the unique least solution. Figure 4.5 shows a portion of the constraint graph generated for the statement `int prod = a*b;`. On the left side, the graph includes one node for each qualifier variable, which is labeled with the corresponding program expression. Each node contains a bit string of length $|Q|$ (not shown in the figure), representing the qualifiers that may be given to the associated expression. All bits are initialized to

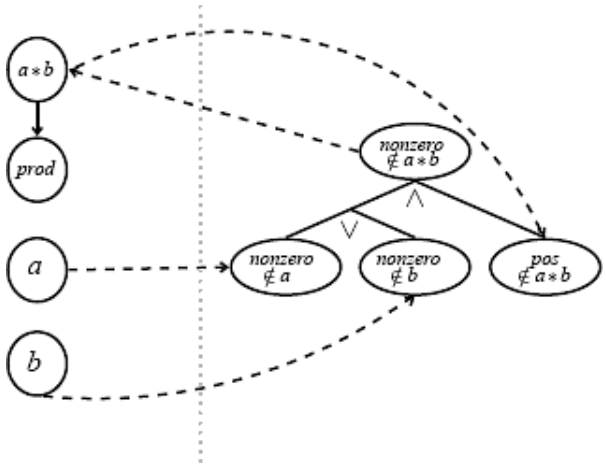

Figure 4.5: An example constraint graph.

true, indicating that all expressions may be given all qualifiers. If bit *i* for node α ever becomes false during constraint solving, this indicates that α cannot include the *i*th qualifier in any solution.

Because this algorithm propagates the *inability* for an expression to have a qualifier, the direction of flow is opposite what one might expect. For each generated constraint of the form $\alpha_1 \supseteq \alpha_2$, the graph includes an edge from α_1 to α_2. For each conditional constraint, the graph contains a representation of its *contrapositive*. For example, the right side of Figure 4.5 shows an *and-or* tree that represents the following constraint:

$$((\texttt{nonzero} \notin \alpha_{\texttt{a}} \vee \texttt{nonzero} \notin \alpha_{\texttt{b}}) \wedge (\mathit{pos} \notin \alpha_{\texttt{a*b}})) \Rightarrow \texttt{nonzero} \notin \alpha_{\texttt{a*b}}$$

The tree's root has an outgoing edge to the `nonzero` bit of the node `a*b`, and the leaves similarly have incoming `nonzero`-bit edges. In the figure, edges to and from individual bits are dotted. The root of each and-or tree maintains a counter of the number of subtrees it is waiting for before it can "fire." This example tree has a counter value of 2.

To solve the constraints, the root of each and-or tree is visited once. If its counter

is greater than 0, CLARITY does nothing. Otherwise, the outgoing edge from its root is traversed, which falsifies the associated bit and propagates this falsehood to its successors recursively until quiescence. For example, if the and-or tree in Figure 4.5 ever fires, that will falsify the `nonzero` bit of `a*b`, which in turn will falsify the `nonzero` bit of `prod`.

After the propagation phase is complete, CLARITY employs the constraints of the form $q \in \alpha$ to check for satisfiability. For each such constraint, if the bit corresponding to qualifier q in node α is false, then this points out a contradiction and the constraints are unsatisfiable. Otherwise, the least solution is formed by mapping each qualifier variable α to the set of all qualifiers whose associated bit in node α is true. For example, the least solution to the constraints in Figure 4.5 is $\alpha_{\texttt{a*b}} = \alpha_{\texttt{prod}} = \{\texttt{nonzero}\}$ and $\alpha_{\texttt{a}} = \alpha_{\texttt{b}} = \{\texttt{nonzero}, \texttt{pos}\}$.

4.4.3.3 Complexity Analysis

Let n be the size of a program, m be the size of the axioms in A, and q be the number of user-defined qualifiers. There are $O(n)$ qualifier variables, $O(n^2)$ constraints of the form $\alpha \supseteq \alpha$, $O(qn)$ constraints of the form $q \in \alpha$, and $O(qn)$ conditional constraints generated, each with size $O(m)$. Therefore, the constraint graph has $O(n^2)$ edges between qualifier-variable nodes, each of which can be propagated across q times. There are $O(qnm)$ edges in total for the and-or trees, and there are $O(qnm)$ edges between the qualifier-variable nodes and the and-or trees, each of which can be propagated across once. Therefore, the total number of propagations, and hence the total time complexity, is $O(qn(n+m))$.

4.5 Automated Qualifier Verification

Up to this point in my discussing of CLARITY, I have focused on defining the language of type qualifier specifications, the semantics of the type systems in the language, and how the qualifiers could be checked and/or inferred. This clearly establishes how CLARITY tackles expressiveness and usability, but does not provide insight on how it addresses *reliability*. In this section, I outline how CLARITY uses the defined *semantic invariant* of the specification to automatically prove that the typing rules are sound using CLARITY's *soundness checker*.

The approach described here is in contrast with the approach shown for JAVACOP in Section 3.5. Because the JAVACOP language is so expressive, reliability is expressed using the incomplete method of unit testing. In the case of CLARITY, the restricted expressiveness of the language allows the type systems to be formally validated against an invariant using an automated theorem prover. Results from such validation provide a much stronger guarantee of soundness in CLARITY than can easily be achieved by hand in JAVACOP. This provides a clear illustration of the trade-offs that are necessary to make practical discipline checking frameworks as defined in Chapter 1.

4.5.1 Soundness Checking

A user-defined qualifier and its associated type rules constitute a typing discipline, which is enforced by the CLARITY typechecker. Often such typing disciplines are intended to ensure a particular run-time invariant. For example, the typing discipline defined by the `pos` qualifier and associated type rules in Listing 4.1 is intended to guarantee that certain expressions only evaluate to positive integers at run time.

However, the extensible typechecker enforces user-defined typing disciplines in a

purely syntactic manner, without knowledge of the intended invariants. For example, suppose the pattern in the second `case` clause in the definition of `pos` in Listing 4.1 were erroneously specified as `E1 - E2` instead of `E1 * E2`. In that case, CLARITY's typechecker would happily use this revised type rule to check programs, even though this can cause `pos`'s intended invariant to be violated at run time.

Rather than forcing users to take responsibility for the correctness of their qualifiers, CLARITY supports *automated soundness checking*. A qualifier definition may optionally specify the qualifier's associated invariant, as discussed in Section 4.2.1.3. The framework's *soundness checker* can then be used to automatically proves that the qualifier's type rules establish this invariant.

With a qualifier's invariant, CLARITY's soundness checker generates one proof obligation for each `case` clause of a qualifier specification and automatically discharges these obligations via Simplify [DNS03], a Nelson-Oppen-style automatic theorem prover [NO79]. Simplify contains decision procedures for several decidable theories, including linear arithmetic and equality for uninterpreted function symbols. Simplify's input language accepts first-order formulas over these theories.

4.5.2 Value Qualifier Proof Obligations

Each `case` clause's proof obligation simply requires that if an expression matches the clause's syntactic pattern and satisfies the clause's predicate, interpreted in the context of an arbitrary run-time execution state ρ, then the qualifier's invariant also holds in ρ. For example, consider the first `case` clause for `pos` in Listing 4.1. The soundness checker generates the following proof obligation: if an expression `E` is an integer constant that is greater than zero, then the value of `E` in an arbitrary execution state ρ is greater than zero. This obligation is easily proved, given the evaluation semantics of integer constants.

Now consider the second `case` clause for `pos`. The soundness checker generates the following proof obligation: if an expression `E` has the form `E1 * E2` and both `E1` and `E2` satisfy `pos`'s invariant in an arbitrary execution state ρ, then `E` also satisfies `pos`'s invariant in ρ. This obligation is easily proved by the semantics of multiplication. On the other hand, if the pattern in that clause were erroneously specified as `E1 - E2`, the soundness checker would catch the error and warn the programmer, since the associated proof obligation would fail: it is not possible to prove that the difference of two arbitrary positive integers is also positive. `Restrict` clauses do not help in determining if an expression of qualified type satisfies its qualifier's invariant, so `restrict` clauses are ignored by the soundness checker.

4.5.2.1 Axioms

CLARITY employs logical axioms to formalize the dynamic semantics of programs in CIL's intermediate language. The state of a program is represented by an execution state $\rho = (\pi, \iota, \varepsilon, \sigma)$, where π is a program, ι is an index pointing to the statement about to be executed, ε is the environment, which maps variable names to memory locations, and σ is the store, which maps locations to values.

CLARITY also defines several function symbols for constructing and manipulating execution states. The *state* function symbol takes a program, index, environment, and store, and it constructs an execution state. The function symbols *getStmt*, *getEnv*, and *getStore* take a state and respectively return the statement about to be executed, the environment, and the store. Environments and stores are represented as *maps*. Simplify has built-in function symbols that represent operations on maps. For example, the built-in *select* function symbol takes a map and a key and returns the key's associated value. Finally, C program expressions and statements themselves are represented using additional function symbols. For instance, the statement $*x := \&y$ is encoded as

$assign(deref(var(x)), addr(var(y)))$.

Given this representation, CLARITY defines axioms for a function symbol *evalExpr*, which evaluates an expression in a given state. For instance, the following axiom formalizes evaluation of variable references

$$\forall \rho, e, x.(e = var(x) \Rightarrow$$
$$evalExpr(\rho, e) = select(getStore(\rho), select(getEnv(\rho), x)))$$

CLARITY similarly defines axioms for a function *location*, which takes an l-value and returns its address, and a function *stepState*, which takes a program state and returns the state resulting from executing the current statement.

The axioms only formalize the subset of the CIL intermediate language necessary for reasoning about expression patterns. For example, CLARITY does not axiomatize the semantics of procedure calls, since they cannot be pattern-matched against. However, CLARITY does explicitly model memory allocation, via the *new* function symbol.

4.5.2.2 Producing Proof Obligations

To produce a qualifier's proof obligations, first CLARITY defines a predicate to represent the qualifier's invariant. Built-in function symbols like `value` in qualifier definitions are translated to their counterpart function symbols in the axioms. For example, the invariant for `pos` from Listing 4.1 is defined as follows:

$$pos(\rho, e) = (evalExpr(\rho, e) > 0)$$

Proving the soundness of a qualifier q also requires access to the invariants of all qualifiers q' that are referred to in q's type rules.

Given these invariants it is straightforward to represent proof obligations in Sim-

plify. For example, the obligation for the second `case` clause of `pos` in Listing 4.1 is defined as follows:

$$\forall \rho, e_1, e_2.(pos(\rho, e1) \wedge pos(\rho, e2)) \Rightarrow pos(\rho, multExpr(e1, e2))$$

4.6 Experience

This section reports on experience using the CLARITY framework for user-defined type qualifiers. I describe experience using the framework to do typechecking on programs both without qualifier inference and with qualifier inference. The experience reported without inference focuses on statically detecting `NULL` dereferences, violations of uniqueness invariants, and improper format strings. These three examples demonstrate all three type qualifier categorizations: value, reference, and flow, respectively. The experience with inference similarly reports on use of a suite of type systems on six open-source C programs. Four of these open-source programs are seen in both sets of experiments.

4.6.1 Qualifier Checking

In all of the experiments described below that do typechecking without inference, the extra compile time for performing qualifier checking in CIL is under one second. Thus, times are omitted from discussion until the next subsection.

4.6.1.1 Null Dereferences

Listing 4.5 shows the definition of a *non-null* value qualifier in CLARITY, which can be automatically proved sound by the CLARITY soundness checker. The sole `case` clause indicates that the address of an l-value can be considered `nonnull`. The `restrict`

```
value qualifier nonnull(T* Expr E)
  case E of
      decl T LValue L:
        &L
  restrict
      decl T* Expr E:
        *E, where nonnull(E)
  invariant value(E) != NULL
```

Listing 4.5: A `nonnull` value qualifier specification in CLARITY.

clause requires all dereferences in a program to be to `nonnull` expressions.

I used this `nonnull` qualifier to statically ensure the absence of `NULL` dereferences in the `grep` search utility program (version 2.5). I annotated the files `dfa.c` and `dfa.h`, which comprise the core string-matching algorithm and related data structures. The files consist of 2287 non-blank, non-comment lines of code.

I applied `nonnull` annotations to variables in an iterative fashion. Running the extensible typechecker on the unannotated files produced an error message for each dereference, due to the `nonnull` qualifier's `restrict` clause. For example, the typechecker indicated a `restrict` violation for `d` in the following code, leading to a `nonnull` annotation:

```
static void build_state(int s, struct dfa *d) {
  ...
  if (d->trcount >= 1024) {
    ...
  }
}
```

These errors were removed by annotating some variables with `nonnull`, which could in turn cause error messages on assignments to the newly-annotated variables, leading to more annotations. In addition to formal parameters and local variables, I documented several fields of structures as being `nonnull` through this process.

Table 4.1: Results from the `nonnull` experiment.

program:	`grep`
files:	`dfa.c`, `dfa.h`
lines:	2287
dereferences:	1072
annotations:	114
casts:	59
errors:	0

There were situations where the type rules for `nonnull` were insufficient and I had to insert casts. The major source of such imprecision is due to the flow-insensitivity of the CLARITY framework. An example of such imprecision from `grep` follows:

```
if ((t = d->trans[works]) != NULL) {
  works = t[*p];
  ...
}
```

The index into array `t` is safe because it is guarded by the check for `NULL`, but the type system cannot deduce this fact. Simple forms of such flow-sensitive reasoning could be incorporated by extending the CLARITY framework with a dataflow analysis framework, such as the one from JAVACOP presented in Section 3.4.

A related source of imprecision occurs when access to a `NULL`-terminated array is guarded by a test that the index is less than the value of a variable holding the array's length. Statically deducing the invariant between the array and that variable may be difficult. One possibility would be to piggyback the qualifier checker on top of CCured [NMW02], which (among other things) can sometimes statically deduce array bounds.

Table 4.1 summarizes the results of this non-null experiment. In order for the

Table 4.2: Results from the `untainted` experiment.

program:	`bftpd`	`mingetty`	`identd`
lines:	750	293	228
`printf` calls:	134	23	21
annotations:	2	1	0
casts:	0	0	0
errors:	1	0	0

`restrict` clause in `nonnull` to succeed on all 1072 dereferences, I had to insert 114 `nonnull` annotations and 59 `nonnull` casts.

4.6.1.2 Untainted Format Strings

While the running examples throughout this dissertation have focused on using an untainted discipline for preventing SQL injection attacks, as mentioned in Section 4.2.1.4, this discipline is general useful for tracking program directives embedded in strings. This section provides experience using the `untainted` qualifier for ensuring proper format-string arguments to `printf`. For these experiments, I used the simple version of `untainted` defined in Section 4.2.1.4, augmented with a `case` clause that defines all constants to be `untainted`:

```
case E of decl T Const C: C
```

This form of the qualifier was used to annotate and check three of the programs tested by Shankar *et al.* [STF01], who performed a taintedness analysis using CQUAL. The programs are `bftpd` (version 1.0.11), an FTP server; `mingetty` (version 0.9.4), a remote terminal utility; and `identd` (version 1.0), a network identification service. For all three programs, I was able to reproduce the results of Shankar *et al.*

These results are shown in table 4.2. Running the `untainted` qualifier checker on `bftpd` indicated two procedure parameters that must be annotated as `untainted`, since they are used as format strings for `printf`. Re-running the qualifier checker then reveals an exploitable error that had been previously identified [Bai, STF01]. The offending code is shown below:

```
int sendstrf(int s, char * untainted format, ...);
...
sendstrf(s, entry->d\_name);
```

The `d_name` field of `entry` is a file name and should not be considered a proper format string. The extensible typechecker appropriately signals an error since the field has not been declared `untainted`.

The other two test programs were verified to have no format-string vulnerabilities. In addition, no casts were required for any of the three test programs; the simple `case` clause defined above was sufficient to infer the untaintedness of all format-string arguments.

4.6.2 Qualifier inference

To test CLARITY's qualifier inference scheme, I used four different qualifier specifications on six open-source C programs, ranging from a few hundred to over 50,000 lines of code: the user identification program `identd`, the terminal program `mingetty`, the regular expression matching program `grep`, the Linux FTP server `bftpd`, the calculator program `bc`, and the intrusion detection system `snort`. Their size and number of variables are shown in the first section of Table 4.3.

Each test case was run through the inferencer twice. The first time, the inferencer was given a definition only for a version of `nonnull`, with a `case` clause indicating that an expression of the form `&E` can be considered `nonnull` and a `restrict` clause

Table 4.3: Qualifier inference results.

qualifier sets			nonnull			nonnull/pos/neg/nz		
program	kloc	vars	cons	gen (s)	solv (s)	cons	gen (s)	solv (s)
identd-1.0	0.19	624	1381	0.09	0.01	2757	0.15	0.01
mingetty-0.9.4	0.21	488	646	0.04	0.01	1204	0.06	0.01
bftpd-1.0.11	2.15	1773	3768	0.39	0.05	6426	0.58	0.08
bc-1.04	4.75	4769	14913	1.21	0.13	27837	5.78	0.18
grep-2.5	10.43	4914	15719	0.75	0.55	28343	7.84	0.71
snort-2.06	52.11	29013	99957	36.39	46.81	176852	290.24	58.07

requiring dereferences to be to `nonnull` expressions, as shown in Listing 4.5. The second time, the inferencer was additionally given versions of the qualifiers `pos`, `neg`, and `nonzero` for integers, each with 5 case rules similar to those in Listings 4.1, 4.3, and 4.5. For each run, the table records the number of constraints produced as well as the time in seconds for constraint generation and constraint solving.

Several pointer dereferences fail to satisfy the `restrict` clause for `nonnull`, causing qualifier inference to signal inconsistencies. I analyzed each of the signaled errors for `bc` and inserted casts to `nonnull` where appropriate to allow inference to succeed. In total, I found no real errors and inserted 107 casts. Of these, 98 were necessary due to a lack of flow-sensitivity in CLARITY; this is the same limitation shown in the tests without inference. Despite this limitation, the qualifier rules were often powerful enough to deduce interesting invariants. For example, on `bc`, 37% (163/446) of the integer lvalues were able to be given the `nonzero` qualifier and 5% (24/446) the `pos` qualifier. For `snort`, 8% (561/7103) of its integer lvalues were able to be given the `nonzero` qualifier, and 7% (317/4571) of its pointer lvalues were able to be given the `nonnull` qualifier (without casts).

As the program size increases, inference becomes too slow to easily use as part of an interactive development process, especially when all four qualifiers are inferred simultaneously. However, at around 6 minutes for a 40,000-line program, it would be feasible to perform global qualifier inference once or occasionally, and use the results to do standard typechecking without inference. This hybrid approach provides the precision and reduced manual annotation requirements of the inference engine with the speed of the standard typechecking engine.

4.7 Summary

In this chapter, I presented the CLARITY framework for semantic type qualifiers in C. The CLARITY language allows developers to create value qualifier specifications via pattern matching over C expressions. While not as expressive as the JAVACOP language, the CLARITY language shows proficiency for developing type qualifiers that constrain integer values such as `nonzero`, `pos`, and `neg`, as well as other simple qualifiers like `untainted` and `nonnull`. This simplified language enables CLARITY to support a number of automated features including standardized qualifier typechecking and inference as well as discipline verification. I presented a formal type system and inference algorithm for qualifiers that clearly defines how qualifiers are enforced on C programs. I also presented the automated soundness checking algorithm that ensures a qualifier's typing rules establish its associated invariant.

CHAPTER 5

Conclusions

In this dissertation, I have shown that syntactic guidelines which I call *programming disciplines* can be used to write better software. This is made possible by taking advantage of three key insights that allow programming disciplines to be *automatically* enforced on target programs via frameworks for programmer-defined discipline checking. First, the frameworks provide a domain-specific language for defining a discipline specification. This allows users to reuse and adapt previously defined disciplines as well as write their own disciplines and encourages enforcement of multiple disciplines on a project. Second, the frameworks leverage static type systems for discipline checking. Type systems often have annotation facilities associated with them that allow users to specify more information about the type of a program element in the source code. The ability to use the built-in annotation facilities, as well as standard idioms of type systems such as subtyping, remove the need to replicate such features in the discipline. The discipline implementations can take advantage of building on top of these underlying base mechanisms, thus significantly simplifying discipline development. Third, the frameworks provide some means by which discipline specifications can be validated against the program invariants they intend to ensure. This third feature elevates discipline enforcement from simply constraint checking to constraint checking for a semantic purpose.

I have developed and presented two practical implementations of this approach: JAVACOP, a pluggable type system for Java; and CLARITY, a semantic type qualifier

framework for C. These implementations demonstrate the usefulness of the approach by showing how a variety of discipline implementations can detect and prevent bugs in real code. However, they also show trade-offs among the desired properties of discipline checking frameworks. JAVACOP has a very expressive language for writing disciplines but has limited faculty for automated verification of disciplines against their intended program invariants. Instead, JAVACOP includes a unit test framework which allows developers to provide test suites looking for violations of a discipline's intent. CLARITY, on the other hand, provides a language with limited expressiveness but more automated verification of the discipline. Limiting the expressiveness of the language also allows CLARITY to provide support for type inference, which is not feasible in JAVACOP.

5.1 Future Directions

While this dissertation has shown the utility of disciplines, there are still a number of interesting directions that it suggests in which research can go towards enabling the development of better software.

One of the key challenges in both formal specification and type systems is automated verification of the soundness. This applies equally well to discipline frameworks shown here. Expressive frameworks like JAVACOP are much more difficult to formally reason about in an automated fashion than simpler systems like CLARITY. However, an increasingly large research focus has shown breakthroughs in both the automatic generation of test suites [BKM02, DDG07] and on model checking of type systems [RHD08]. Such work suggests that it might be possible to provide more automated support for verifying disciplines in more expressive frameworks like JAVACOP. The translation scheme from JAVACOP to Datalog$^\neg$ is a first step in this direction as it should allow more formal reasoning about the disciplines themselves.

As more programming migrates to platforms that support interaction of multiple languages, discipline frameworks that focus on disciplines for a single target language like CLARITY and JAVACOP will not be sufficient for discipline checking. Instead, new models must be investigated to find the appropriate balance between language-agnostic and language-specific discipline specification. Some possible platforms to target with this work include web platform languages like JavaScript, SQL, and PHP; and languages that run on top of the .NET Common Language Runtime like C# and F#. Recent work in this area includes type safety checking over foreign function calls [FF08] and the Boogie language which multiple languages can be translated into and verified with the Boogie tool [DL05, BCD05].

APPENDIX A

Full FJCOP to Datalog$^\neg$ Translation Scheme

In this appendix, I present the full translation scheme from FJCOP to Datalog$^\neg$. Discussion of the translation can be found in Chapter 3, Section 3.3. This full translation includes a number of translation scheme judgments that were not found in previous discussion, including alternate rules for effectively handling negation, multiple literals instead of a single literal created via translation

Value Expression $\rightarrow^v$

[TRANS-CONSTANT]

$$\frac{}{\Phi \vdash \ell \Rightarrow^v \ell \mid \ell \dashv \Phi}$$

[TRANS-VAR-KNOWN]

$$\frac{\Phi(x) = X}{\Phi \vdash x \Rightarrow^v X \mid X \dashv \Phi}$$

[TRANS-VAR-FRESH]

$$\frac{\Phi(x) = \emptyset \qquad X \text{ fresh}}{\Phi \vdash x \Rightarrow^v X \mid X \dashv \Phi \cup \{(x, X)\}}$$

[TRANS-VALUE-FN]

$$\frac{\Phi \vdash t \Rightarrow^v T \mid T \dashv \Phi_0 \qquad \forall_{i=1}^{|\bar{t}|}.\Phi_{i-1} \vdash t_i \Rightarrow^v T_i \mid T_i \dashv \Phi_i \qquad X \text{ fresh}}{\Phi \vdash t.f_v(\bar{t}) \Rightarrow^v f_v(T, \overline{T}, X) \mid X \dashv \Phi_{|\bar{t}|}}$$

Boolean Expressions $\Rightarrow^b$

[TRANS-PRED-CALL]

$$\frac{\forall_{i=1}^{|\bar{t}|}.\Phi_{i-1} \vdash e_i \Rightarrow^v T_i \mid T_i \dashv \Phi_i}{\Phi_0 \vdash f(\bar{t}) \Rightarrow^b f(\overline{T}) \dashv \Phi_{|\bar{t}|}}$$

[TRANS-BINDING-TYPE-TEST-1]

$$\frac{\Phi \vdash e_v \Rightarrow^v E \mid T \dashv \Phi' \qquad E \neq T}{\Phi \vdash x : \tau \leftarrow e_v \Rightarrow^b E, \mathtt{type}(T,\tau) \dashv \Phi' \cup \{(x,T)\}}$$

[TRANS-BINDING-TYPE-TEST-2]

$$\frac{\Phi \vdash e_v \Rightarrow^v T \mid T \dashv \Phi'}{\Phi \vdash x : \tau \leftarrow e_v \Rightarrow^b \mathtt{type}(T,\tau) \dashv \Phi' \cup \{(x,T)\}}$$

[TRANS-NEG-1]

$$\frac{\Phi \vdash e_b \Rightarrow^b \overline{E}, E \dashv \Phi' \qquad E \neq \neg(E')}{\Phi \vdash \neg e_b \Rightarrow^b \overline{E}, \neg(E) \dashv \Phi'}$$

vspace.1in

[TRANS-NEG-2]

$$\frac{\Phi \vdash e_b \Rightarrow^b \overline{E}, \neg(E) \dashv \Phi'}{\Phi \vdash \neg e_b \Rightarrow^b \overline{E}, E \dashv \Phi'}$$

[TRANS-BOOL-FN]

$$\frac{\Phi \vdash t \Rightarrow^v \Phi_0 \vdash T \mid T \qquad \forall_{i=1}^{|\bar{t}|}.\Phi_{i-1} \vdash t_i \Rightarrow^v T_i \mid T_i \dashv \Phi_i}{\Phi \vdash t.\mathcal{F}_b(\bar{t}) \Rightarrow^b \mathcal{F}_b(T,\overline{T}) \dashv \Phi_{|\bar{t}|}}$$

Constraints $\Rightarrow^c$

[TRANS-SEQUENCE]

$$\frac{\Phi \vdash c_1 \Rightarrow^c \overline{E_1} \mid \overline{C_1} \qquad \Phi \vdash c_2 \Rightarrow^c \overline{E_2} \mid \overline{C_2}}{\Phi \vdash c_1; c_2 \Rightarrow^c \overline{E_1}, \overline{E_2} \mid \overline{C_1} \cup \overline{C_2}}$$

[TRANS-REQUIRE]

$$\frac{\Phi \vdash e_b \Rightarrow^b \overline{E} \dashv \Phi'}{\Phi \vdash \mathbf{require}(e_b) \Rightarrow^c \overline{E} \mid \emptyset}$$

[TRANS-WHERE-1]

$$\frac{F \text{ fresh} \qquad \overline{T} = \text{range}(\Phi) \qquad \Phi \vdash e_b \Rightarrow^b E \dashv \Phi' \qquad \Phi' \vdash c \Rightarrow^c \overline{E_c} \mid \overline{C}}{\Phi \vdash \textbf{where}(e_b)\{c\} \Rightarrow^c F(\overline{T}) \mid \left\{ \begin{array}{l} F(\overline{T}) :- E, \overline{E_c}. \\ F(\overline{T}) :- \bigwedge_{i=1}^{|\overline{T}|} \texttt{object}(T_i), \neg(E). \end{array} \right\} \cup \overline{C}}$$

[TRANS-WHERE-2]

$$\frac{F \text{ fresh} \qquad \overline{T} = \text{range}(\Phi) \qquad \Phi \vdash e_b \Rightarrow^b E_1, E_2 \dashv \Phi' \qquad \Phi' \vdash c \Rightarrow^c \overline{E_c} \mid \overline{C}}{\Phi \vdash \textbf{where}(e_b)\{c\} \Rightarrow^c F(\overline{T}) \mid \left\{ \begin{array}{l} F(\overline{T}) :- E_1, E_2, \overline{E_c}. \\ F(\overline{T}) :- \bigwedge_{i=1}^{|\overline{T}|} \texttt{object}(T_i), E_1, \neg(E_2). \end{array} \right\} \cup \overline{C}}$$

[TRANS-FORALL-LIST]

$$\frac{\begin{array}{c} F, X_1, L, L' \text{ fresh} \\ \Phi \vdash x_2 \Rightarrow^v T_2 \mid T_2 \dashv \Phi \qquad \overline{T} = \text{range}(\Phi) \qquad \Phi \cup \{(x_1, X_1)\} \vdash c \Rightarrow^c \overline{E} \mid \overline{C} \end{array}}{\begin{array}{c} \Phi \vdash \forall_{\mathcal{L}} x_1 : \tau \in x_2.\{c\} \Rightarrow^c \\ F(\overline{T}, T_2) \mid \left\{ \begin{array}{l} F(\overline{T}, L) :- \texttt{cons}(X_1, L', L), \texttt{type}(X_1, \tau), \overline{E}, F(\overline{T}, L'). \\ F(\overline{T}, L) :- \texttt{cons}(X_1, L', L), \neg(\texttt{type}(X_1, \tau)), F(\overline{T}, L'). \\ F(\overline{T}, L) :- \texttt{nil}(L). \end{array} \right\} \cup \overline{C} \end{array}}$$

[TRANS-FORALL-TREE]

$$\frac{\begin{array}{c} F, X_1, L, L' \text{ fresh} \\ \Phi \vdash x_2 \Rightarrow^v T_2 \mid T_2 \dashv \Phi \qquad \overline{T} = \text{range}(\Phi) \qquad \Phi \cup \{(x_1, X_1)\} \vdash c \Rightarrow^c \overline{E} \mid \overline{C} \end{array}}{\begin{array}{c} \Phi \vdash \forall_{\mathcal{A}} x_1 : \tau \in x_2.\{c\} \Rightarrow^c \texttt{subnodes}(T_2, L), F(\overline{T}, L) \mid \\ \left\{ \begin{array}{l} F(\overline{T}, L) :- \texttt{cons}(X_1, L', L), \texttt{type}(X_1, \tau), \overline{E}, F(\overline{T}, L'). \\ F(\overline{T}, L) :- \texttt{cons}(X_1, L', L), \neg(\texttt{type}(X_1, \tau)), F(\overline{T}, L'). \\ F(\overline{T}, L) :- \texttt{nil}(L). \end{array} \right\} \cup \overline{C} \end{array}}$$

[TRANS-EXISTS-LIST]

$$\frac{F, X_1, L, L' \text{ fresh} \qquad \Phi \vdash x_2 \Rightarrow^v T_2 \mid T_2 \dashv \Phi \qquad \overline{T} = \text{range}(\Phi) \qquad \Phi \cup \{(x_1, X_1)\} \vdash c \Rightarrow^c \overline{E} \mid \overline{C}}{\Phi \vdash \exists_{\mathcal{L}} x_1 : \tau \in x_2.\{c\} \Rightarrow^c F(\overline{T}, T_2) \mid \left\{ \begin{array}{l} F(\overline{T}, L) :- \mathtt{cons}(X_1, L', L), \mathtt{type}(X_1, \tau), \overline{E}. \\ F(\overline{T}, L) :- \mathtt{cons}(X_1, L', L), F(\overline{T}, L'). \end{array} \right\} \cup \overline{C}}$$

[TRANS-EXISTS-LIST]

$$\frac{F, X_1, L, L' \text{ fresh} \qquad \Phi \vdash x_2 \Rightarrow^v T_2 \mid T_2 \dashv \Phi \qquad \overline{T} = \text{range}(\Phi) \qquad \Phi \cup \{(x_1, X_1)\} \vdash c \Rightarrow^c \overline{E} \mid \overline{C}}{\Phi \vdash \exists_{\mathcal{L}} x_1 : \tau \in x_2.\{c\} \Rightarrow^c \mathtt{subnodes}(T_2, L), F(\overline{T}, L) \mid \left\{ \begin{array}{l} F(\overline{T}, L) :- \mathtt{cons}(X_1, L', L), \mathtt{type}(X_1, \tau), \overline{E}. \\ F(\overline{T}, L) :- \mathtt{cons}(X_1, L', L), F(\overline{T}, L'). \end{array} \right\} \cup \overline{C}}$$

Predicate and Rule Definitions $\Rightarrow^d$

[TRANS-PREDICATE]

$$\frac{\Phi_0 = \emptyset \qquad \forall_{i=1}^{|\overline{x}|}.\Phi_{i-1} \vdash x_i \Rightarrow^v X_i \mid X_i \dashv \Phi_i \qquad \Phi_{|\overline{x}|} \vdash c \Rightarrow^c \overline{E} \mid \overline{C}}{\mathbf{declare}\ f(\overline{x} : \overline{\tau})\{c\} \Rightarrow^d \{f(\overline{X}) :- \bigwedge_{i=1}^{|\overline{x}|} \mathtt{type}(X_i, \tau_i), \overline{E}.\} \cup \overline{C} \mid \emptyset}$$

[TRANS-RULE]

$$\frac{\emptyset \vdash x \Rightarrow^v X \mid X \dashv \Phi \qquad \Phi \vdash c \Rightarrow^c \overline{E} \mid \overline{C}}{\mathbf{rule}\ f(x : \tau)\{c\} \Rightarrow^d \{f(X) :- \mathtt{type}(X, \tau), \overline{E}.\} \cup \overline{C} \mid \{? :- \mathtt{type}(X, \tau), \mathtt{not}(f(X)).\}}$$

Program $\Rightarrow^p$

[TRANS-PROGRAM]

$$\frac{\forall_{i=1}^{|\overline{d}|}.d_i \Rightarrow^d \overline{C_i} \mid G_i}{\overline{d} \Rightarrow^p \bigcup_{i=1}^{|\overline{d}|} \overline{C_i} \mid \bigcup_{i=1}^{|\overline{d}|} G_i}$$

REFERENCES

[AKC02] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. "Alias annotations for program understanding." In Mamdouh Ibrahim and Satoshi Matsuoka, editors, *OOPSLA 2002: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Seattle, WA, USA, November 2002, pp. 311–330, New York, NY, 2002. ACM Press.

[ANM06] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. "A framework for implementing pluggable type systems." In Tarr and Cook [TC06], pp. 57–74.

[Bai] Christophe Bailleux. "More security problems in bftpd-1.0.12. `BugTraq` mailing list post of December 8, 2000." `http://www.securityfocus.com/archive/1/149977`.

[BBD00] Greg Bollela, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[BCD05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs 0002, and K. Rustan M. Leino. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs." In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO 2005: Formal Methods for Components and Objects, 4th International Symposium*, Amsterdam, The Netherlands, November 2005, Revised Lectures, volume 4111 of *Lecture Notes in Computer Science*, pp. 364–387. Springer, 2005.

[BDF08] Mike Barnett, Robert Deline, Manuel Fähndrich, Bart Jacobs, K. Rustan Leino, Wolfram Schulte, and Herman Venter. "The Spec# Programming System: Challenges and Directions." pp. 144–152, 2008.

[BE04] Adrian Birka and Michael D. Ernst. "A practical type system and language for reference immutability." In Vlissides and Schmidt [VS04], pp. 35–49.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. "Korat: automated testing based on Java predicates." In Phyllis G. Frankl, editor, *ISSTA 2002: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, Roma, Italy, July 2002, pp. 123–133, New York, NY, USA, 2002. ACM Press.

[BLC02] E. Bruneton, R. Lenglet, and T. Coupaye. "ASM: A Java bytecode manipulation and analysis framework." In *Proceedings of Adaptable and Extensible Component Systems*, Grenoble, France, November 2002, 2002. `http://asm.objectweb.org`.

[Blo02] J. Bloch. "A metadata facility for the Java programming language." Technical Report JSR 175, `http://www.jcp.org`, 2002.

[Bok99] Boris Bokowski. "CoffeeStrainer: statically-checked constraints on the definition and use of types in Java." In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 355–374. Springer-Verlag, 1999.

[Boy01] John Boyland. "Alias burying: Unique variables without destructive reads." *Software - Practice and Experience*, **31**(6):533–553, 2001.

[C02] "C# Language Specification, Second Edition. ECMA International, Standard ECMA-334.", December 2002.

[CAB86] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[CGT90] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Springer-Verlag, Berlin, Germany, 1990.

[CH88] Thierry Coquand and Gerard Huet. "The calculus of constructions." *Inf. Comput.*, **76**(2-3):95–120, 1988.

[CJ07] Patrice Chalin and Perry R. James. "Non-null References by Default in Java: Alleviating the Nullity Annotation Burden." In Erik Ernst, editor, *ECOOP 2007: Proceedings of the 21st European Conference on Object-Oriented Programming, Berlin, Germany, July 2007*, volume 4609 of *Lecture Notes in Computer Science*, pp. 227–247, Berlin, Germany, 2007. Springer.

[CMM04] Brian Chin, Shane Markstrum, and Todd Millstein. "Semantic Type Qualifiers." Technical Report CSD-TR-40045, UCLA Computer Science Department, November 2004.

[CMM05a] Brian Chin, Shane Markstrum, and Todd Millstein. "Semantic Type Qualifiers." In Vivek Sarkar and Mary W. Hall, editors, *PLDI 2005: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, IL, USA, June 2005, pp. 85–95, New York, NY, 2005. ACM Press.

[CMM05b] Brian Chin, Shane Markstrum, Todd Millstein, and Jens Pal sberg. "Inference of User-Defined Type Qualifiers and Qualifier Rules." Technical Report CSD-TR-050041, UCLA Computer Science Department, October 2005.

[CMM06] Brian Chin, Shane Markstrum, Todd Millstein, and Jens Palsberg. "Inference of User-Defined Type Qualifiers and Qualifier Rules." In Peter Sestoft, editor, *ESOP 2006: Programming Languages and Systems - Proceedings of the 15th European Symposium on Programming*, Vienna, Austria, March 2006, volume 3924 of *Lecture Notes in Computer Science*, pp. 264–278, Berlin, Germany, 2006. Springer.

[Cop05] Tom Copeland. *PMD Applied*. Centennial Books, November 2005.

[Cre97] Roger F. Crew. "ASTLOG: A Language for Examining Abstract Syntax Trees." In Chris Ramming, editor, *Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, California, USA, October 1997, pp. 229–243, Berkeley, CA, 1997. USENIX Association.

[CV02] Karl Crary and Joseph C. Vanderwaart. "An expressive, scalable type theory for certified code." In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pp. 191–205. ACM Press, 2002.

[DDG07] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. "Automated Testing of Refactoring Engines." In Ivica Crnkovic and Antonia Bertolino, editors, *ESEC/SIGSOFT FSE 2007: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Dubrovnik, Croatia, September 3-7, 2007, pp. 185–194, New York, NY, 2007. ACM Press.

[DeM04] Linda DeMichiel. *Enterprise JavaBeans Specification, Version 3.0*. SUN Microsystems, 2004.

[DF01] Robert DeLine and Manuel Fahndrich. "Enforcing high-level protocols in low-level software." In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pp. 59–69. ACM Press, 2001.

[DF04] Robert DeLine and Manuel Fahndrich. “Typestates for Objects.” In *Proceedings of the 2004 European Conference on Object-Oriented Programming*, LNCS 3086, Oslo, Norway, June 2004. Springer-Verlag.

[DL05] Robert DeLine and K. Rustan M. Leino. “BoogiePL: A typed procedural language for checking object-oriented programs.” Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[DMR92] Carolyn K. Duby, Scott Meyers, and Steven P. Reiss. “CCEL: A Metalanguage for C++.” In *Proceedings of the 1992 USENIX C++ Conference*, Portland, OR, USA, August 1992, pp. 99–116, Berkeley, CA, 1992. USENIX Association.

[DNS03] David Detlefs, Greg Nelson, and James B. Saxe. “Simplify: A Theorem Prover for Program Checking.” Technical Report HPL-2003-148, HP Labs, 2003.

[ECG01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. “Dynamically Discovering Likely Program Invariants to Support Program Evolution.” *IEEE Transactions on Software Engineering*, **27**(2):99–123, 2001.

[EH04] Torbjörn Ekman and Görel Hedin. “Rewritable Reference Attributed Grammars.” In Martin Odersky, editor, *ECOOP 2004: Proceedings of the 18th European Conference on Object-Oriented Programming*, Oslo, Norway, June 2004, volume 3086 of *Lecture Notes in Computer Science*, pp. 144–169, Berlin, Germany, 2004. Springer.

[EMO04] Michael Eichberg, Mira Mezini, Klaus Ostermann, and Thorsten Schäfer. “XIRC: A Kernel for Cross-Artifact Information Engineering in Software Development Environments.” In *Working Conference on Reverse Engineering*, 2004.

[Ern07] Michael Ernst. “Java Annotations on Types.” Technical Report JSR 308, `http://www.jcp.org`, 2007.

[ESM05] Michael Eichberg, Thorsten Schäfer, and Mira Mezini. “Using Annotations to Check Structural Properties of Classes.” In Maura Cerioli, editor, *FASE 2005: Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering*, Edinburgh, UK, April 2005, volume 3442 of *Lecture Notes in Computer Science*, pp. 237–252, Berlin, Germany, 2005. Springer.

[FF00] Cormac Flanagan and Stephen N. Freund. "Type-based race detection for Java." In James Larus and Monica Lam, editors, *PLDI 2000: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Vancouver, BC, Canada, June 2000, pp. 219–232, New York, NY, 2000. ACM Press.

[FF08] Michael Furr and Jeffrey S. Foster. "Checking type safety of foreign function calls." *ACM Trans. Program. Lang. Syst.*, **30**(4):1–63, 2008.

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. "A Theory of Type Qualifiers." In Barbara G. Ryder and Benjamin Zorn, editors, *PLDI 1999: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, Atlanta, GA, USA, June 1999, pp. 192–203, New York, NY, 1999. ACM Press.

[FL03] Manuel Fahndrich and K. Rustan M. Leino. "Declaring and checking non-null types in an object-oriented language." In Ron Crocker and Guy L. Steele Jr., editors, *OOPSLA 2003: Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, Anaheim, CA, USA, October 2003, pp. 302–312, New York, NY, 2003. ACM Press.

[FLL02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. "Extended Static Checking for Java." In Jens Knoop and Laurie J. Hendren, editors, *PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002, pp. 234–245, New York, NY, 2002. ACM Press.

[FM07] Jeffery S. Foster and Kin-keung Ma. "Inferring Aliasing and Encapsulation Properties for Java." In Gabriel et al. [GBL07], pp. 423–440.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. "Flow-sensitive type qualifiers." In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pp. 1–12. ACM Press, 2002.

[GBL07] Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors. *OOPSLA 2007: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Montreal, Quebec, Canada, October 2007, New York, NY, 2007. ACM Press.

[GF07] David Greenfieldboyce and Jeffery S. Foster. “Type Qualifier Inference for Java.” In Gabriel et al. [GBL07], pp. 321–336.

[GHJ95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.

[GM05] Joseph (Yossi) Gil and Itay Maman. “Micro Patterns in Java Code.” In Johnson and Gabriel [JG05], pp. 97–116.

[GMJ02] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. “Region-based memory management in Cyclone.” In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pp. 282–293. ACM Press, 2002.

[HH06] Daqing Hou and H. James Hoover. “Using SCL to Specify and Check Design Intent in Source Code.” *IEEE Transactions on Software Engineering*, **32**(6):404–423, 2006.

[HL02] Sudheendra Hangal and Monica S. Lam. “Tracking down software bugs using automatic anomaly detection.” In Michal Young and Jeff Magee, editors, *ICSE 2002: Proceedings of the 22rd International Conference on Software Engineering*, Orlando, Florida, USA, May 2002, pp. 291–301, New York, NY, USA, 2002. ACM Press.

[Hoa69] C. A. R. Hoare. “An axiomatic basis for computer programming.” *Commun. ACM*, **12**(10):576–580, 1969.

[Hog91] John Hogg. “Islands: aliasing protection in object-oriented languages.” In Andreas Paepcke, editor, *OOPSLA 1991: Proceedings of the 6th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Phoenix, AZ, USA, November 1991*, pp. 271–285, New York, NY, 1991. ACM Press.

[HP04] David Hovemeyer and William Pugh. “Finding bugs is easy.” In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA Companion 2004: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, BC, Canada, October 2004, pp. 132–136, New York, NY, 2004. ACM Press.

[HVM06] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. “CodeQuest: Scalable Source Code Queries with Datalog.” In Dave Thomas, editor, *ECOOP 2006: Proceedings of the 20th European Conference on*

Object-Oriented Programming, Nantes, France, July 2006, volume 4067 of *Lecture Notes in Computer Science*, pp. 2–27, Berlin, Germany, 2006. Springer.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. "Featherweight Java: a minimal core calculus for Java and GJ." *ACM Transactions on Programming Languages and Systems*, **23**(3):396–450, May 2001.

[ISO] ISO/IEC 25436:2006. *Information Technology – Eiffel: Analysis, Design and Programming Language*. ISO, Geneva, Switzerland.

[JG05] Ralph E. Johnson and Richard P. Gabriel, editors. *OOPSLA 2005: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Diego, CA, USA, October 2005, New York, NY, 2005. ACM Press.

[Jt00] "JUnit." `http://junit.org`, 2000.

[JV03] Doug Janzen and Kris De Volder. "Navigating and querying code without getting lost." In William G. Griswold and Mehmet Aksit, editors, *AOSD 2003: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, Boston, Massachusetts, USA, March 2003, pp. 178–187, New York, NY, 2003. ACM Press.

[JW04] Rob Johnson and David Wagner. "Finding User/Kernel Pointer Bugs with Type Inference." In *Proceedings of the 13th USENIX Security Symposium*, pp. 119–134, 2004.

[LLL99] Gary T. Leavens, Gary T. Leavens, Gary T. Leavens, Albert L. Baker, Albert L. Baker, and Albert L. Baker. "Enhancing the Pre- and Post-condition Technique for More Expressive Specifications." In *In FM'99: World Congress on Formal Methods*, pp. 1087–1106. Springer, 1999.

[LMC03] Sorin Lerner, Todd Millstein, and Craig Chambers. "Automatically proving the correctness of compiler optimizations." In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pp. 220–231. ACM Press, 2003.

[LMR05] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. "Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules." In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2005.

[Mar82] Per Martin-Löf. "Constructive Mathematics and Computer Programming." In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pp. 153–175, Amsterdam, 1982. North-Holland.

[Mey92] Bertrand Meyer. "Applying "Design by Contract"." *Computer*, **25**(10):40–51, 1992.

[Mil04] Todd Millstein. "Practical predicate dispatch." In Vlissides and Schmidt [VS04], pp. 345–364.

[MME] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. "JavaCOP Declarative Pluggable Types for Java." to be published.

[MWC99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. "From System F to typed assembly language." *ACM Trans. Program. Lang. Syst.*, **21**(3):527–568, 1999.

[Mye99] Andrew C. Myers. "JFlow: Practical mostly-static information flow control." In Andrew W. Appel and Alex Aiken, editors, *POPL 1999: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Antonio, TX, USA, January 1999, pp. 228–241, New York, NY, 1999. ACM Press.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. "Polyglot: An Extensible Compiler Framework for Java." In Görel Hedin, editor, *CC 2003: Proceedings of the 12th International Conference on Compiler Construction*, Warsaw, Poland, April 2003, volume 2622 of *Lecture Notes in Computer Science*, pp. 138–152, Berlin, Germany, 2003. Springer.

[Nec97] George C. Necula. "Proof-carrying code." In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 106–119. ACM Press, 1997.

[NMR02] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs." In *Proceedings of CC 2002: 11'th International Conference on Compiler Construction*. Springer-Verlag, April 2002.

[NMW02] George C. Necula, Scott McPeak, and Westley Weimer. "CCured: Type-safe retrofitting of legacy code." In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 128–139. ACM Press, 2002.

[NO79] Greg Nelson and Derek C. Oppen. “Simplification by Cooperating Decision Procedures.” *ACM Trans. Program. Lang. Syst.*, **1**(2):245–257, 1979.

[NQM06] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. “J&: Software Composition with Nested Intersection.” In Tarr and Cook [TC06], pp. 21–36.

[PAC08] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. “Practical pluggable types for Java.” In Barbara G. Ryder and Andreas Zeller, editors, *ISSTA 2008: Proceedings of the 2008 ACM/SIGSOFT International Symposium on Software Testing and Analysis*, Seattle, WA, USA, July 2008, pp. 201–212, New York, NY, 2008. ACM Press.

[PNC06] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. “Generic Ownership for Generic Java.” In Tarr and Cook [TC06], pp. 311–324.

[Pt04] “Polyglot extensible compiler framework.” `http://www.cs.cornell.edu/Projects/polyglot`, 2004.

[Pt07] “Polyglot for Java 5.” `http://www.cs.ucla.edu/~milanst/projects/polyglot5`, 2007.

[Rem98] Didier Rémy and Jérôme Vouillon. “Objective ML: An Effective Object-Oriented Extension of ML.” *Theory and Practice of Object Systems*, **4**(1):27–52, 1998.

[35 references.]

[RHD08] Michael Roberson, Melanie Harries, Paul T. Darga, and Chandrasekhar Boyapati. “Efficient software model checking of soundness of type systems.” In Gail E. Harris, editor, *OOPSLA 2008: Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Nashville, TN, USA, October 2008, pp. 493–504, New York, NY, 2008. ACM Press.

[RTC92] RTCA. “Software considerations in airborne systems and equipment certification.”, 1992.

[SCJ] “JSR 302: Safety Critical Java Technology.” `http://jcp.org/en/jsr/detail?id=302`.

[SST02] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. “A type system for certified binaries.” In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 217–232. ACM Press, 2002.

[STF01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. "Detecting Format String Vulnerabilities with Type Qualifiers." In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., August 2001.

[TC06] Peri L. Tarr and William R. Cook, editors. *OOPSLA 2006: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Portland, OR, USA, October 2006, New York, NY, 2006. ACM Press.

[TE05] Matthew S. Tschantz and Michael D. Ernst. "Javari: adding reference immutability to Java." In Johnson and Gabriel [JG05], pp. 211–230.

[US 09] US-CERT. "SQL Injection.", 2009.

[VB99] Jan Vitek and Boris Bokowski. "Confined types." In Brent Hailpern, Linda Northrop, and A. Michael Berman, editors, *OOPSLA 1999: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Denver, CO, USA, November 1999*, pp. 82–96, New York, NY, 1999. ACM Press.

[VS04] John M. Vlissides and Douglas C. Schmidt, editors. *OOPSLA 2004: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, BC, Canada, October 2004, New York, NY, 2004. ACM Press.

[WL04] John Whaley and Monica S. Lam. "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams." In William Pugh and Craig Chambers, editors, *PLDI 2004: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, Washington, DC, USA, June 2004, pp. 131–144, New York, NY, 2004. ACM Press.

[WSM06] Alessandro Warth, Milan Stanojević, and Todd Millstein. "Statically scoped object adaptation with expanders." In Tarr and Cook [TC06], pp. 37–56.

[XP98] Hongwei Xi and Frank Pfenning. "Eliminating Array Bound Checking through Dependent Types." In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 249–257, Montreal, June 1998.

[XP99] Hongwei Xi and Frank Pfenning. "Dependent Types in Practical Programming." In *Proceedings of the 26th ACM SIGPLAN Symposium on*

Principles of Programming Languages, pp. 214–227, San Antonio, January 1999.

[ZNV04] Tian Zhao, James Noble, and Jan Vitek. "Scoped Types for Real-Time Java." In *RTSS Proceedings*, 2004.