Science of Computer Programming 23(1):19–53, 1994.

Static Typing for Object-Oriented Programming

Jens Palsberg
palsberg@daimi.aau.dk

Michael I. Schwartzbach
mis@daimi.aau.dk

Computer Science Department
Aarhus University
Ny Munkegade
DK-8000 Århus C, Denmark

Abstract

We develop a theory of statically typed object-oriented languages. It represents classes as labeled, regular trees, types as finite sets of classes, and subclassing as a partial order on trees. We show that our subclassing order strictly generalizes inheritance, and that a novel genericity mechanism arises as an order-theoretic complement. This mechanism, called class substitution, is pragmatically useful and can be implemented efficiently.

1 Introduction

Object-oriented programming is becoming widespread. Numerous programming languages supporting object-oriented concepts are in use, and theories about object-oriented design and implementation are being developed and applied [21, 3, 14].

An important issue in object-oriented programming is to obtain *reusable* software components [30]. This is achieved through the notions of *object*, *class*, *inheritance*, *late binding*, and the imperative constructs of *variables* and *assignments*. Such features are found for example in the SMALLTALK language [25] in which a large number of reusable classes have been written. SMALLTALK, however, is untyped. Though this is ideal for prototyping and exploratory development, a static type system is required to ensure that programs are readable, reliable, and efficient.

This paper studies an idealized subset of SMALLTALK, equipped with a novel static type system. Our analysis results in the definition of a new genericity mechanism, called class substitution, which we prove to be the order-theoretic complement of inheritance.

In the following section we briefly review the benefits of static typing. We argue that the choice of *finite sets of classes* as types can provide exactly those benefits. Furthermore, in this setting subtyping can be seen to be simply *set inclusion*. Finally, we compare this notion of type with others that have been proposed.

In Section 3 we outline the example language. It is an idealized subset of SMALLTALK, in which variables and parameters are declared together with types. The language has been simplified by the omission of blocks; instead, a primitive if-then-else is provided. We also give precise requirements for static correctness of programs, and introduce a mathematical framework in which we represent classes as *labeled, regular trees*. These representations abstract away from class names.

In Section 4 we discuss inheritance and its interaction with mutually recursive classes. We also show that a program using inheritance can be transformed into one which does not.

In Section 5 we structure the class representations with a partial order which generalizes inheritance. The intuition behind the order is that a subclass may extend the implementation and redefine the types consistently, while preserving the recursive structure of the superclass. All such generalized subclass relationships can be exploited by an implementation to provide reusable software components. The suggested implementation is a generalization of a naïve SMALLTALK interpreter. We show that the partial order has the same characteristic properties as its subset inheritance: it is decidable, has a least element, has finite intervals, does not allow temporal cycles, and preserves subtyping.

In Section 6 we prove that the generalized subclassing order is strictly more powerful than ordinary inheritance. We characterize the extra possibilities by showing that they form a suborder which is an order-theoretic orthogonal complement to the suborder formed by inheritance relationships.

In Section 7 we develop the orthogonal complement of inheritance into a programming mechanism, called class substitution, which turns out to be a genericity mechanism. This means that our simple type system, though voided of e.g. type variables, still supports genericity. We extend the example language with syntax for class substitution, give example programs, and compare with parameterized classes.

Finally, in Section 8 we use our framework to analyze the kind of type system which is common in existing object-oriented languages, for example C++ and SIMULA/BETA. The analysis yields an explanation of why these languages often allow loop-holes in the type system or resort to run-time type-checking in some cases.

2 Types

Types are advantageous because an untyped program may be unreadable, unreliable, and inefficient. Any choice of type system for a language must be able to remedy some or all of the above deficiencies [29].

Types may be used as annotations, and those can be read not only by humans but also by the compiler which may be able to exhibit a safety-guarantee and perform compile-time optimizations. The safety-guarantee will typically state that operations are only performed on arguments of the proper types; in other words, certain run-time errors will not occur.

In this section we present a new, simple type system for object-oriented languages and we argue why it yields the benefits stated above. We also examine other type systems and discuss similarities and differences.

2.1 Types are Sets of Classes

```
class Record
    var key: Integer
    method getKey returns Integer
        key
    method setKey(k: Integer) returns Record
        key := k ; self
end Record

class File
    var buffer: Record
    method initialize returns File
        buffer := Record new ; buffer.setKey(17) ; self
end File
```

Figure 1: Records and Files.

The basic metaphor in object-oriented programming is the *object*. An object groups together variables and procedures (called methods), and prevents direct outside access to the variables; it may be thought of as a module [58]. Objects are instances of classes, see for example figure 1. The class Record specifies a pattern from which all Record objects will be created. Such an object is created for example in class File by the expression "Record new" and it gets a separate copy of all variables. Note that also Integer is a class, though specified elsewhere, and that each method returns the result of its last expression. If nothing needs to be returned, then usually one returns the object itself, denoted by self.

The only possible interaction with an object is when sending a message to it—when invoking one of its methods. For example, in class File the expression buffer.setKey(17) expresses the sending of the message setKey with argument 17 to the object in the variable buffer. If the object does not understand the message—if its class does not implement a method with the specified name—then the run-time error messageNotUnderstood occurs. In a *pure* object-oriented language this is the *only* run-time error that can occur.

The purpose of a type system is to allow the programmer to annotate programs with information about which methods are available in a given object, and to allow the compiler to guarantee the the error messageNotUnderstood will never occur [4, 9]. The latter immediately enables compile-time optimizations because a number of checks need not be inserted into the code.

In SMALLTALK, any object can be assigned to any variable. Message sending is implemented by *late binding*, i.e., the message send is dynamically bound to an implementation depending on the class of the receiver. The fundamental question is: which messages will an object residing in a variable be able to respond to? Ignoring the possibility of doing flow analysis, the answer is: those methods that are common to the classes of the possible objects in that variable. This set of methods can be specified by simply listing the classes of the possible objects, because only finitely many classes occur in a given program. These observations lead us to define the notion of type that will be analyzed throughout this paper.

A type is a finite set of classes.

Note that a type can only contain classes that are part of the program. This corresponds to a "closed-world" assumption.

Our types are more general than they may seem at first. Any kind of type expressions may be employed, providing that they can be interpreted as predicates on classes. In a given program only finitely many classes will satisfy any given predicate. The type can now be identified with the finite set; hence, we are justified in viewing our type system as being quite general.

As an example of a type, consider again figure 1, where class File specifies a variable buffer of type Record. The type contains a single class, hence, only Record objects are allowed to reside in the variable.

2.2 Subtyping is Set Inclusion

An object may have more than one type. Consider for example an instance of class Record. The singleton type containing just Record is a type of that instance, but so is also any superset. Henceforth, we will call a superset for a *supertype*, and a subset for a *subtype*.

When type-checking an assignment x := e (and similarly for a parameter passing), then it suffices to check that the static type of x is a supertype of the static type of e. This is because the assignment then will not violate the requirement that only objects of the static type of x can reside in x.

If we had a more advanced kind of type expressions, then subtyping must be defined to coincide with—or at least to respect—inclusion of the corresponding sets.

The nil object requires special attention. In Smalltalk it is an instance of the class undefinedObject, and it is the initial contents of variables when objects are created. This implies that we should want undefinedObject to be a member of all types. In this paper, we do not want to rely on any predefined classes, however, so we will treat nil in another way. Instead of having undefinedObject as an explicit member of all types, we will define nil to be a special primitive value which implicitly has the empty type and, hence, has all types. Treating nil as a non-instance is in line with its implementation in typed languages such as C++ [54], Eiffel [39], and Simula [22]/Beta [32]. In the language we later present, nil is in fact the only constant value. The language can be viewed as simply a calculus of pointers, in which it is quite natural that nil should enjoy a special status.

2.3 Inheritance is Not Subtyping

Object-oriented programming differs from other programming paradigms in offering inheritance as a means for reusing class definitions. Inheritance can be viewed as a shorthand: it allows the definition of a class to be a modification of an existing one. More specifically, it allows the construction of *subclasses* by adding variables and methods, and by overriding method bodies [25, 57]. A thorough discussion of inheritance is given in a later section.

The difference between inheritance and subtyping is illustrated in figure 2 which displays a class hierarchy discussed in [36], together with the corresponding type hierarchy. Notice that we turn the class hierarchy "upside-down" in order to get the smallest class at the bottom, and that the figure uses the notation $\uparrow$ C for the set of all subclasses of C, even potential ones. The class hierarchy is a *tree*, whereas the type hierarchy is a *lattice*.

Note that the Beta group has suggested interpreting nil as an instance of an auxiliary class on *top* of the class hierarchy [36]. This is awkward because it implies that this class can be obtained by some sort of multiple inheritance of all other classes. This again implies that instances of this auxiliary class should be able to respond to any message; clearly nil is *not* able to do this. Our explanation of nil as a value having the empty type is more satisfactory: it reflects that nil can *not* respond to all messages, and that it can be assigned to any variable. In

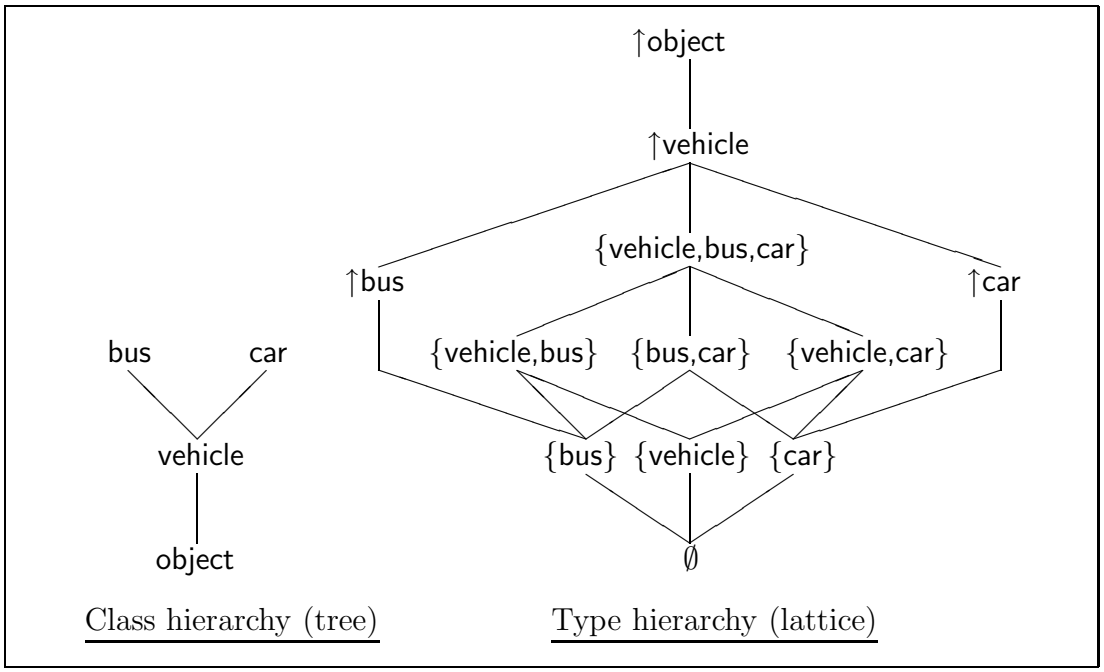

Figure 2: The class and type hierarchies (excerpts).

the language we later present, nil "understands" only the "if-then-else" message.

2.4 Other Type Systems

Usually, formal models of typed object-oriented programming are based on the lambda calculus. They represent objects as records and methods as functions, and involve coercions together with subtypes [10, 41], polymorphic types [40, 13, 23], or F-bounded constraints [8, 15, 7] in the description of inheritance. In contrast, traditional object-oriented languages are not based on coercions and do not support methods as values.

Furthermore, the coercion models—while being very general in some respects—do not support variables and assignments, because variable (mutable) types have no non-trivial subtypes, as observed by Cardelli [11]. In a functional language, an assignment must be emulated by the creation of an updated copy, and it is extremely hard to preserve the type of the original value. It was long believed that bounded parametric polymorphism was sufficient [13], but it has been realized that considerably more fine-grained type systems are required to handle even simple updates [12].

Graver and Johnson's type system for SMALLTALK [29, 27, 28] has much in common with ours. Their types are essentially finite sets of classes, but they have to *axiomatize* a subtype relation that corresponds exactly to set inclusion because

the type system involves type variables. They also employ a notion of *signature type* which essentially denotes the finite set of subclasses of a given class, under our "closed-world" assumption.

One strength of our type system is that it can avoid type variables and exclusively use the simple notion of sets of classes as types. The issue of *genericity* will instead be handled by generalizing the notion of subclassing.

3 The Example Language

(Program)	P	::=	$C_1 \ldots C_n$ E
(Class)	C	::=	**class** ClassId **var** $D_1 \ldots D_k$ $M_1 \ldots M_n$ **end** ClassId
(Method)	M	::=	**method** MethodId ($D_1 \ldots D_k$) returns T E
(Declaration)	D	::=	Id : T
(Type)	T	::=	[selfClass] $\text{ClassId}_1 \ldots \text{ClassId}_n$
(Expression)	E	::=	Id := E | E . MethodId ($E_1 \ldots E_n$) | E ; E | if E then E else E | ClassId new | selfClass new | E instanceOf ClassId | self | Id | nil

Figure 3: Syntax of the example language.

Our example language is an idealized subset of SMALLTALK, except that variables and parameters are declared together with a type, see figure 3. We also leave the issue of inheritance to the following section. In this section we briefly discuss the semantics of the language and state the precise rules for static correctness. We also introduce a convenient representation of classes.

3.1 Informal Semantics

A *program* is a set of classes followed by an expression whose value is the result of executing the program. A *class* can contains variables and methods; a *method* consists of a name, an argument-list, a result-type, and an expression. The result of an assignment is the assigned value. A message send is executed by evaluating the receiver and the arguments, and if the class of the receiver implements a

method for the message, then a normal procedure call takes place; otherwise, the error messageNotUnderstood occurs. The result of a sequence is the result of the last expression in that sequence. The if-then-else expression tests if the condition is non-nil. The expression "selfClass new" yields an instance of the class of self. The expression "E instanceOf ClassId" yields a run-time check for class membership. If the check fails, then the expression evaluates to nil. The expression self denotes the receiver of the message, Id refers to instance variables and parameters, and nil is a primitive value.

Note that selfClass can always be replaced by the name of the enclosing class. This is a convenient way of making recursion explicit. It is not sufficient for expressing mutually recursive classes, however. For that, it is necessary to use the class names directly.

We have no primitive types (like integer and boolean) nor type constructors (like list) because we can program classes that encode them, see [44].

Note that we ignore the issues of concurrency and persistence [56].

3.2 Static Correctness

We define correctness of a method body with respect to the name of the enclosing class, and global and local environments. These can be uniquely determined from the program syntax.

The global environment maps class names to class descriptions. A class description maps method names to method descriptions. A method description maps argument numbers to their declared types; for convenience, the result type is assumed to have number zero.

The local environment is a finite map from names of instance variables and parameters to their declared types.

We shall use the notation

$$\mathsf{E} :: \tau$$

to denote that the expression E is statically correct and has type τ. This property will be defined by means of a number of rules of the form

- Condition_1
  $\vdots$
  Condition_k
  $\boxed{\mathsf{E} :: \tau}$

with the obvious interpretation: if all the conditions can be satisfied, then the conclusion follows.

The following rules exhaust the syntactic possibilities; the enclosing class has name C, the global environment is denoted by $\mathcal{G}$, and the local environment is denoted by $\mathcal{E}$.

- $\boxed{\mathsf{nil} :: \{\}}$
- $\boxed{\mathsf{self} :: \{\mathsf{C}\}}$
- $\boxed{\mathsf{selfClass\ new} :: \{\mathsf{C}\}}$
- $\boxed{\mathsf{D\ new} :: \{\mathsf{D}\}}$
- $\mathsf{id} \in dom(\mathcal{E})$ — early check
  $\boxed{\mathsf{Id} :: \mathcal{E}(\mathsf{Id})}$
- $\mathsf{E} :: \tau$
  $\boxed{\mathsf{E\ instanceOf\ D} :: \{\mathsf{D}\}}$
- $\mathsf{E}_i :: \tau_i$
  $\boxed{\mathsf{if\ E_1\ then\ E_2\ else\ E_3} :: \tau_2 \cup \tau_3}$
- $\mathsf{E}_i :: \tau_i$
  $\boxed{\mathsf{E_1\ ;\ E_2} :: \tau_2}$
- $\mathsf{E} :: \tau_1$
  $\mathsf{Id} :: \tau_2$
  $\tau_1 \subseteq \tau_2$ — subtype check
  $\boxed{\mathsf{Id{:=}E} :: \tau_1}$
- $\mathsf{E} :: \tau$
  $\mathsf{E}_i :: \tau_i$
  $\forall c \in \tau :\ \mathsf{m} \in dom(\mathcal{G}\, c)$ — early check
  $\forall c \in \tau :\ dom(\mathcal{G}\, c\, \mathsf{m}) = \{0, \ldots, n\}$ — early check
  $\forall c \in \tau :\ \tau_i \subseteq \mathcal{G}\, c\, \mathsf{m}\, i$ — subtype check
  $\boxed{\mathsf{E\ .\ m(E_1\ \ldots\ E_n)} :: \bigcup_{c \in \tau} \mathcal{G}\, c\, \mathsf{m}\, 0}$

Now, a program is statically correct when all method bodies have a typing in the corresponding environments.

Note that really only two kinds of checks are performed:

- *early checks*, which require the existence of certain declared names.
- *subtype checks*, which require inclusions between certain types.

With this definition of static correctness, it should be obvious that variables of type T can only contain objects of type T; that if an expression is evaluated to a non-nil result, then the class of that result is contained in the type of the expression; and that we can guarantee that any message is sent to either nil or an instance of a class which implements a method for that message. Note that we ignore keeping track of nil-values; this can be treated separately by flow analysis.

A formal proof of these claims should be based on a dynamic semantics of the example language [48, 31]. We will not go into the details in this paper, however, but move on to define a convenient representation of classes.

3.3 Tree Representations of Classes

We shall work with a slightly abstracted form of classes, which will allow us to give a formal treatment of their relationships.

The mathematical framework is a kind of labeled trees, which we shall call L-trees.

Definition 3.1: Let Σ be a finite alphabet. An *L-tree* over Σ is an ordered, node-labeled, regular tree. We recall that a tree is *regular* when it has only finitely many *different* subtrees [20]. The labels are finite strings over $\Sigma \cup \{\bullet\}$, where it is assumed that $\bullet \notin \Sigma$. The empty label is denoted by Ω. We shall refer to the special symbol $\bullet$ as a *gap*. It is further required that any node in an L-tree has exactly one subtree for each gap in its label. $\Box$

We have some notation associated with such trees.

Definition 3.2: Let T be an L-tree. A *tree address* is a sequence of integers denoting a path from the root to a node; each integer denotes the number of a subtree in the given order. We write $\alpha \in T$ when α is a valid tree address in T. The empty tree address is denoted by λ. If $\alpha \in T$ then $T[\alpha]$ denotes the label with address α in T, and $T \downarrow \alpha$ denotes the subtree of T whose root has address α. Note that $T \downarrow \lambda = T$. $\Box$

We can now separate a class from its surrounding program and represent it as an

L-tree. Intuitively, the tree is a possibly infinite unfolding of the source code of the class.

The labels will be source code with gaps in place of occurrences of class names. Recall that a class name may occur before new, after instanceOf, and inside type expressions. We shall also replace occurrences of selfClass with gaps.

The root label of a class representation is the gapped source code of the class. Let the classes in the program be $\mathsf{C}_1,\ldots,\mathsf{C}_n$ and the corresponding root labels be $L_1,\ldots,L_n$. The trees $\textsc{tree}(\mathsf{C}_1),\ldots,\textsc{tree}(\mathsf{C}_n)$ are defined through the following regular equation system:

$$\begin{array}{rcl} \textsc{tree}(\mathsf{C}_1) & = & L_1(\bar{X}_1) \\ & \vdots & \\ \textsc{tree}(\mathsf{C}_n) & = & L_n(\bar{X}_n) \end{array}$$

Each $\bar{X}_i$ is a list of subtrees, which is obtained as follows. Every subtree corresponds to a gap in the label L_i. If the gap replaced the class C_j, then the subtree is $\textsc{tree}(\mathsf{C}_j)$; if the gap replaced selfClass, then the subtree is $\textsc{tree}(\mathsf{C}_i)$. It is well-known that such an equation system has a unique solution [19], which clearly is an L-tree. If any part of a class is recursive, then the tree will be infinite.

Quite often, recursive types are represented as regular trees with nodes labeled by type constructors [2]. This is in fact what we have done, with the proviso that we consider every class as a user-defined type *constructor*—rather than as a user-defined type.

We have now abstracted away from the class *names*, which obviously cannot be uniquely recovered. We can, however, transform a tree T back into an *equivalent* program, $\textsc{prog}(T)$, by selecting a class name for every different subtree in T and reconstructing the syntax. The structural equivalence on classes simply says that two classes are equivalent if their trees are equal.

It might be considered that the trees are not sufficiently abstract to deserve attention. For instance, we distinguish the order in which methods are implemented, and type expressions are interpreted as sequences rather than sets. Some tidying up is certainly possible: methods could be ordered in a canonical way, and type expressions could be normalized. For the purposes of this paper, however, such improvements are not really necessary. We shall mainly look at classes obtained as *modifications* of other classes, in which particular arbitrary choices are always carried along. The *behavior* of a class, for example an element of a Scott-domain, is clearly not a feasible representation—it is too abstract.

We can now define a *universe* of classes as follows:

$$\mathcal{U} = \{\textsc{tree}(\mathsf{C}) \mid \mathsf{C} \text{ is a class in some statically correct program}\}$$

This is a mathematical object on which we later shall observe an interesting structure.

3.4 Examples

```
class Object
end Object

class A
    var x: Object
end A

class B
    var x: Object
    method set(y: Object) returns selfClass
        x := y ; self
end B
```

Figure 4: Example classes.

We now provide some short examples of the above definitions. Consider the classes A and B in figure 4. Let us first consider what the corresponding trees look like. We shall use the abbreviations

$$L_{\mathsf{A}} = \textbf{var } \mathsf{x}\text{: } \bullet$$
$$L_{\mathsf{B}} = \textbf{var } \mathsf{x}\text{: } \bullet \textbf{ method } \mathsf{set}(\mathsf{y}\text{: } \bullet) \textbf{ returns } \bullet \ \mathsf{x} := \mathsf{y} \ ; \ \mathsf{self}$$

The equation system for the trees is now

$$\textsc{tree}(\mathsf{Object}) = \Omega()$$
$$\textsc{tree}(\mathsf{A}) = L_{\mathsf{A}}(\textsc{tree}(\mathsf{Object}))$$
$$\textsc{tree}(\mathsf{B}) = L_{\mathsf{B}}(\textsc{tree}(\mathsf{Object}), \textsc{tree}(\mathsf{Object}), \textsc{tree}(\mathsf{B}))$$

The corresponding trees are pictured in figure 5. We can finally observe that

$$\textsc{tree}(\mathsf{B})[3,3,2] = \Omega$$
$$\textsc{tree}(\mathsf{B})[3,3,3] = L_{\mathsf{B}}$$

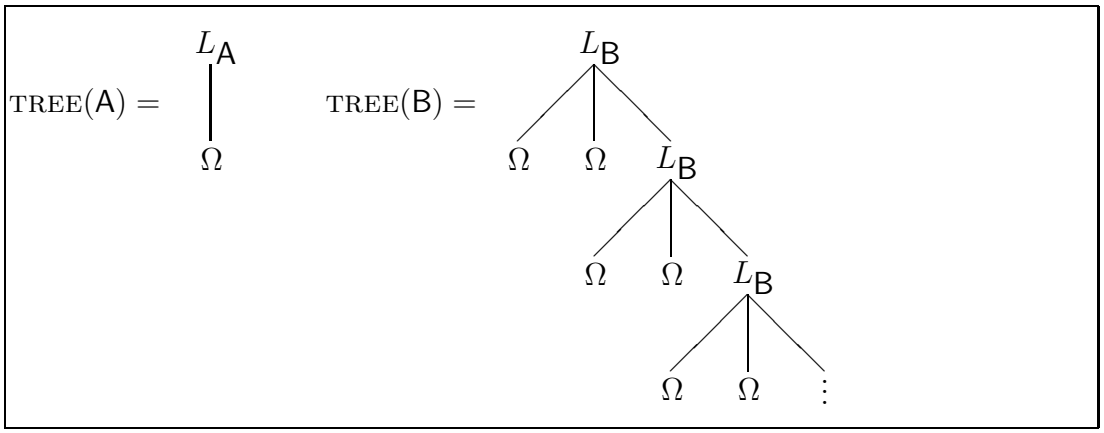

Figure 5: Example trees.

4 Inheritance

Inheritance is reuse of class definitions. It may significantly increase programmer productivity and decrease source code size. In this section we add inheritance to our example language, discuss its properties, and show that we can use the universe of class representations from the previous section to represent also the classes defined by inheritance.

4.1 Syntax

To introduce inheritance we extend the grammar in figure 3 as showed in figure 6.

(Class) C	::=	**class** ClassId **inherits** ClassId **var** $D_1 \ldots D_k$ $M_1 \ldots M_n$ **end** ClassId

Figure 6: Syntax of Inheritance.

The class name following **inherits** is called the *superclass* of the class being defined; the latter is called a *subclass* of the superclass.

The subclass is a modification of the superclass: it may add variables and methods, and if a method name coincides with an existing one, then the new method definition overrides the old one. The body of a new method definition may refer to both the existing variables and the existing methods. It has been argued by Snyder [53] that much better encapsulation is achieved if only the existing methods can be referred; we ignore this consideration in this paper. For a denotational semantics of inheritance, see [16, 48, 31, 17].

4.2 Properties

Inheritance can be used in various ways, ranging from undisciplined code-grabbing to disciplined program structuring based on a hierarchical design method [21, 3, 14, 30]. Common to all approaches is that the superclass is created *before* the subclass. We will henceforth use the terminology that the subclass is *temporally dependent* on its superclass.

A class can be a subclass of another class which itself is defined by inheritance. The chain of superclasses is always finite, however, because the program is finite. Also, the superclass chain will contain no cycles; we will say that it is *temporally acyclic*. Note that any non-empty class can be defined as a subclass of the empty class, which we will call Object. This is actually enforced in SMALLTALK. In this situation, the inheritance hierarchy is a tree, otherwise it is a forest.

If C is the superclass and D is the subclass, then it is common to say that D IS-A C [6, 55]. For example, if Student is a subclass of Person, then it seems reasonable to say that Student IS-A Person. It is convenient to let IS-A denote the transitive closure of this relation. The other possible relation between classes is HAS-A. If a class C declares a variable of a type which contains the class D, then we will say that C HAS-A D. For example, if the Student declares a graduate variable of type Boolean, then Student HAS-A Boolean. We will also say that C HAS-A D if C mentions D in a parameter list or as an argument of new or instanceOf. Analogously with IS-A, we let HAS-A denote the transitive closure of this relation.

It is crucial not to confuse IS-A and HAS-A. If C HAS-A D and D HAS-A C, then we will say that C and D are mutually recursive. In comparison, it is impossible to have both C IS-A D and D IS-A C because that would be a temporal cycle. To make any sense, mutually recursive classes must be defined simultaneously by the programmer.

Together, IS-A and HAS-A impose a temporal order on the classes in a program. The intuition is that a class D depends temporally on a class C if C has to be created before D. Formally, we can represent a program as a directed graph. The graph has one node for every class definition in the program. Each node has a label which is the gapped source code of the corresponding implementation. There will be two kinds of edges: IS-A and HAS-A. We have an IS-A edge from every subclass to its immediate superclass, and a HAS-A edge from every gap in the label to the node representing the missing class. We can then define *temporal dependency* to be a path between two nodes containing at least one IS-A edge. A temporal dependency from D to C means that C must be created before D.

If the classes in a program contains no temporal cycles, then we will call it *well-formed*. A well-formed program can be transformed into one which does not use inheritance, as demonstrated below.

Our reason for doing this transformation is that when all classes are represented as elements of our universe of class representations, then it makes sense to analyze this universe in order to *discover* other relations between classes besides IS-A and HAS-A. Such relations will be independent both of class names and of the particular shorthands that can be used in program texts. So far, the only shorthand we have encountered is inheritance, but later on we will define another called class substitution. Pedersen [47] proposed the notion of generalization which is the inverse of inheritance.

The loss of an explicit class hierarchy may at first seem to cause severe problems, since some programming mechanisms depend on exactly this. In particular, we think about *redefinition* of method bodies in subclasses, and about the constructs super [25] and inner [33, 37]. However, these mechanisms depend primarily on the existence of multiple implementations of methods. This we can certainly handle, since a label contains a sequence of implementations of methods—several of which may have the same name. The dynamic behavior of a message send is to search the label from *right to left*, and to execute the first implementation of the method that is found. This gives the correct semantics of method redefinitions. The construct super can be viewed as a directive to search from the location of the present method implementation towards the left. Dually, the construct inner directs the search to go from the present location towards the right. This will nearly give the usual semantics, and is certainly in line with the explanation given in [5].

When expanding the inheritance shorthand it is necessary to be careful when encountering recursive occurrences of a class. A class C is recursive if C HAS-A C. If a class D inherits this class C, then after expansion, all occurrences of C must have been transformed into D. The reason is that a variable of a type which contains C could be assigned to a variable which contains selfClass. Since selfClass always denotes the class it appears in, it will automatically denote D after the expansion. Hence, C must be transformed into D to preserve static correctness. The complications get worse when considering mutually recursive classes; the algorithm in the following subsection gives a detailed solution to the general case.

While it is often the natural choice to transform all recursive occurrences during inheritance, one can certainly find program examples where the opposite choice is preferable. However, with our structural equivalence on classes recursive occurrences *must* be transformed in order to make the subclass statically correct. The problem could be solved by introducing *opacity* operators on classes, in line with [43]. In this paper we will not explore this aspect further.

4.3 The Expansion Algorithm

We now present the algorithm that expands a program using inheritance into an equivalent one that does not. The idea behind the algorithm is simple: it rewrites the program in the same fashion that a programmer would have to if the class should be written in a language that does not support inheritance.

Consider an inheritance specification

B **inherits** A

where the classes A and B (except the "**inherits** A" part) are represented by the variables A_1 and B_1 in the following minimized regular equation systems:

$$\begin{array}{rclcrcl} A_1 & = & L_1(\bar{A}) & & B_1 & = & K_1(\bar{B}) \\ & \vdots & & & & \vdots & \\ A_n & = & L_n(\bar{A}) & & B_m & = & K_m(\bar{B}) \end{array}$$

Then the expanded version of B is represented by the variable Z in the (not necessarily minimized) regular equation system obtained as follows. Form the union of the equation systems for A and B, remove the equations for A_1 and B_1, add the equation

$$Z = L_1 K_1(\bar{A}, \bar{B})$$

and finally rename all occurrences of A_1 and B_1 into Z.

```
class A
    var x: A
end A

class B inherits A
    var y: A
    var z: B
end B
```

Figure 7: Before expansion of inheritance.

This process is illustrated by the following simple example. Consider the classes in figure 7. The minimized regular equations are as follows:

$$\begin{array}{rcl} A_1 & = & (\textbf{var x: } \bullet)(A_1) \\ B_1 & = & (\textbf{var y: } \bullet \textbf{ var z: } \bullet)(B_2, B_1) \\ B_2 & = & (\textbf{var x: } \bullet)(B_2) \end{array}$$

According to the algorithm, the regular equations for the expanded class are

$$\begin{aligned} B_2 &= (\textbf{var } \mathsf{x}\text{: } \bullet)(B_2) \\ Z &= (\textbf{var } \mathsf{x}\text{: } \bullet \textbf{ var } \mathsf{y}\text{: } \bullet \textbf{ var } \mathsf{z}\text{: } \bullet)(Z, B_2, Z) \end{aligned}$$

which translates into the class in figure 8. Note how the recursion at x has been captured.

```
class A
    var x: A
end A

class B
    var x: B
    var y: A
    var z: B
end B
```

Figure 8: After expansion of inheritance.

With this expansion algorithm at hand, we can now consider expanding programs where more than one class is defined using inheritance. Given that a program contains no temporal cycles, the classes can be sorted in temporal order and then expanded one by one. For a detailed explanation and examples of this, see [46].

5 Generalized Subclassing

The main purpose of providing an independent notion of *classes* is to define a generalized, structural notion of *subclassing*. This arises directly from the application on $\mathcal{U}$ of a partial order on general L-trees.

5.1 A Generalized Interpreter

In [45] we present a generalization of a naïve SMALLTALK interpreter. This interpreter supports inheritance through a *method lookup*—a run-time search for implementations of methods. Our extended interpreter also does a run-time search for arguments to new and instanceOf operations. This allows a more general form of code reuse, which we can express through a partial order $\lhd$ on classes. In [45] we give a precise description of the run-time environments of the extended interpreter, and we show the following property: if $T_1 \lhd T_2$ holds, then any run-time implementation of T_1 can be *extended* to yield a run-time implementation

of T_2. The code reuse that can be expressed through inheritance corresponds to a *suborder* of $\lhd$.

In this section we shall define the partial order $\lhd$ and show a number of its formal properties. In the following sections we shall develop a programming mechanism that is complementary to inheritance; as we shall see, their combination realizes all of $\lhd$.

5.2 A Partial Order on Trees

Intuitively, the relation $\lhd$ imposes three different constraints on subclasses. Each of these reflect that the subclass *reuses* the implementation of the superclass.

- *the labels may only be extended:* this simply means that the subclass can only extend the implementation and not modify existing parts. This also ensures that all early checks will remain satisfied.
- *equal classes must remain equal:* this ensures that all subtype checks will remain satisfied; hence, the code of the superclass can only be reused in a manner that preserves static correctness.
- *the recursive structure must be preserved:* this is essential for allowing the code to be reused since different code is generated for selfClass and other classes [45].

The partial order $\lhd$ is our generalized notion of subclassing, such that if A is the superclass and B is the subclass, then $A \lhd B$. It may seem strange that *super* is smaller than *sub*, but this is a common confusion of terminology. Clearly, the subclass has a larger implementation than the superclass; equally clearly, the superclass is more general than the subclass. We choose to retain the usual terminology, while employing the mathematically suggestive ordering $\lhd$.

To be able to define $\lhd$ formally, we introduce some auxiliary concepts. The first is a simple partial order on L-trees.

Definition 5.1: The usual prefix order on finite strings is written as $\leq$. The partial order $T_1 \sqsubseteq T_2$ on L-trees over Σ holds exactly when

- $\forall \alpha \in T_1 : \alpha \in T_2$
- $\forall \alpha \in T_1 : T_1[\alpha] \leq T_2[\alpha]$

Note that $\sqsubseteq$ is the *node-wise* extension of $\leq$. $\square$

The order $\sqsubseteq$ reflects that labels may only be extended. We next provide a simple, finite representation of an L-tree.

Proposition 5.2: Every L-tree T can be represented by a finite, partial, deterministic automaton with labeled states, with language $\{\alpha \mid \alpha \in T\}$, and where α is accepted in a state labeled $T[\alpha]$.
Proof: The finitely many different subtrees all become accept states with the label of their root. The transitions of the automaton are determined by the fan-out from the corresponding root. $\square$

These automata provide finite representations of L-trees. The idea of representing a regular tree as an automaton is also exploited in [51, 52]. All later algorithms will in reality work on such automata.

Proposition 5.3: The partial order $\sqsubseteq$ is decidable.
Proof: The algorithm is a variation of the standard one for language inclusion on the corresponding automata. $\square$

The second auxiliary concept is the notion of a generator for an L-tree.

Definition 5.4: If T is an L-tree over Σ, then its *generator* $\textsc{gen}(T)$ is another L-tree which is obtained from T by replacing all maximal, proper occurrences of T itself by a singleton tree with the special label $\diamond$; it is assumed that $\diamond$ is incomparable with all other labels in the $\leq$-ordering. We say that T is *recursive* when $T \neq \textsc{gen}(T)$. Note that the generator itself may be an infinite tree, and that the original tree can readily be recovered from its generator. $\square$

The generator of a class makes explicit all the recursive occurrences of the class itself. For example, all occurrences of selfClass in its source code are replaced by $\diamond$, but also mutual recursion is captured.

We are now ready to define $\lhd$ using the order $\preceq$ which is a subset of $\sqsubseteq$.

Definition 5.5: The relation $T_1 \preceq T_2$ on L-trees is the largest subset of $\sqsubseteq$ such that the following *stability* condition holds

- $\forall \alpha, \beta \in T_1 :\ T_1 \downarrow \alpha = T_1 \downarrow \beta \ \Rightarrow\ T_2 \downarrow \alpha = T_2 \downarrow \beta$

The relation $T_1 \lhd T_2$ on L-trees holds exactly when

- $\forall \alpha \in T_1 :\ \textsc{gen}(T_1 \downarrow \alpha) \preceq \textsc{gen}(T_2 \downarrow \alpha)$

Note that if $T_1 \lhd T_2$ then for any $\alpha \in T_1$ we also have $T_1 \downarrow \alpha \lhd T_2 \downarrow \alpha$. $\square$

Since $\preceq$ is a subset of $\sqsubseteq$, it reflects that labels may only be extended. Furthermore, the stability condition ensures that equal classes remain equal. The relation $\lhd$ is then defined so that the generators at all levels are in the $\preceq$ relation. This ensures that the recursive structure is preserved.

Proposition 5.6: The relations $\preceq$ and $\lhd$ are decidable, partial orders.
Proof: Clearly, $\preceq$ is a partial order since stability is reflexive and transitive; also, $\lhd$ is a partial order because $\preceq$ is.

Since by proposition 5.3 we know that $\sqsubseteq$ is decidable, we must only show that stability is, too. On *minimized* automata, representing the trees T_1 and T_2, stability translates to the property that any two words α, β accepted in the same state by the T_1-automata must also be accepted in the same state by the T_2-automata. This property can be decided for general automata using a simple, linear-time dynamic programming algorithm.

To decide $\lhd$ we can rely on decidability of $\preceq$ and the fact that L-trees have only finitely many different subtrees, all of which can easily be constructed. $\square$

5.3 Properties

The subclassing order $\lhd$ has the same characteristic properties as inheritance: it has a least element, has finite intervals, does not allow temporal cycles, and preserves subtyping. In this subsection we prove these claims.

Proposition 5.7: The partial order $\lhd$ has a least element $\bot$.

Proof: Clearly, $\perp$ is just the singleton tree with the label Ω. $\square$

In a class hierarchy, $\perp$ corresponds to the empty class Object. To show that $\lhd$ has finite intervals, we need a notion of unfolding directed graphs.

Definition 5.8: Let G be a directed, rooted graph containing a path from the root to each vertex. A particular unfolding of G, called UNFOLD(G), is obtained by the following variation of the standard depth-first search algorithm [1] starting in the root. The modification is that if the current edge leads to a previously visited vertex in a *different* strongly connected component, then a fresh copy of that entire component is inserted in the graph. See for example figure 9, where (v_1, v_3) is the current edge. The graph UNFOLD(G) can be understood as a tree of possibly multiple copies of the strongly connected components of G. $\square$

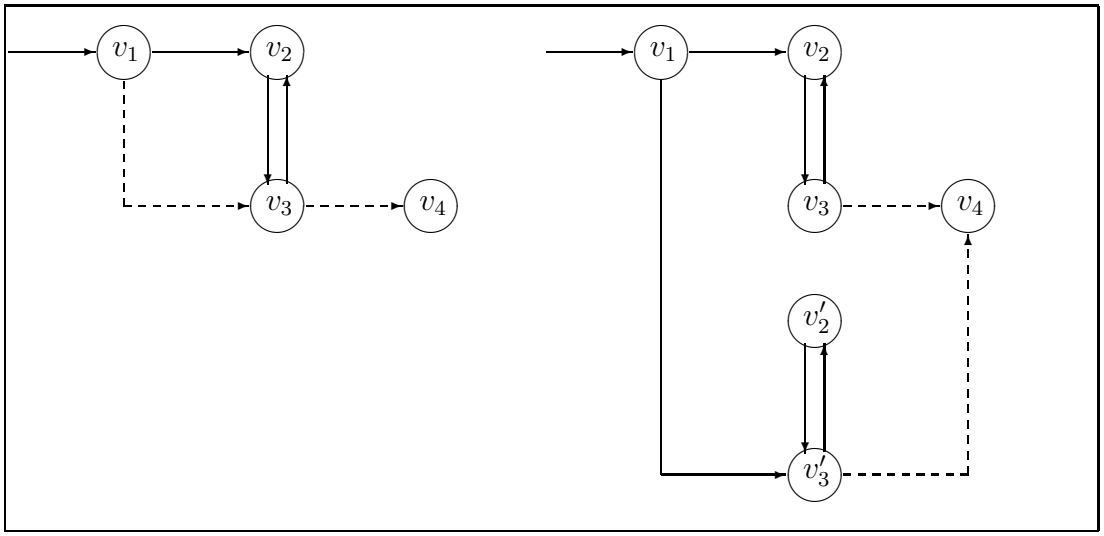

Figure 9: A step in the construction of UNFOLD(G).

Lemma 5.9: A depth-first traversal of UNFOLD(G) has the property that if the current edge leads to a previously visited vertex, then that vertex is on a cycle of already processed edges and the current edge.
Proof: Let (v, w) be the current edge. If we have previously visited w, then, by construction of UNFOLD(G), v and w are in the same strongly connected component. Because of the depth-first strategy, there is a path of already processed

edges from w to v. The result follows. $\Box$

Proposition 5.10: For any L-tree T_2 the interval $\{T_1 \mid T_1 \triangleleft T_2\}$ is finite.
Proof: For the purposes of this proof, we shall represent L-trees by their *canonical* automata. This is obtained by subjecting the minimal automaton to the unfolding described in definition 5.8. Clearly, this new automaton will have the same language and represent the same L-tree; in particular, the L-tree can be recovered from the automaton.

Now, assume that $T_1 \triangleleft T_2$. Let A_1 and A_2 be their canonical automata. We shall construct a total function h from states of A_1 to states of A_2 with the following properties

- h maps the initial state of A_1 to that of A_2
- if $x \xrightarrow{i} y$ is a transition in A_1, then $h(x) \xrightarrow{i} h(y)$ is a transition in A_2
- the label of x is $\leq$ that of $h(x)$
- h is injective

The construction works iteratively through a depth-first traversal of A_1. At any stage the current h will satisfy all of the above properties, but it may be partial. We start with just the pair of initial states, which is clearly legal.

We proceed by examining the current unexplored depth-first A_1-transition $x \xrightarrow{i} y$ from a state x in the domain of h. This is matched by an A_2-transition $h(x) \xrightarrow{i} z$, since the label of x is $\leq$ than that of $h(x)$. The function h is now extended to $h' = h \cup \{y \mapsto z\}$. Only two necessary properties are not immediate: that h' is still a function, and that h' is still injective.

Assume that we have already seen y before; we must assure that $z = h(y)$. Having seen y before means, from lemma 5.9, that we have a cycle from y to y. Now look at the generator of the subtree of T_1 that corresponds to y. The cycle that we have traversed is here a path from the root to a $\Diamond$-label. In the $h(y)$-generator of T_2 the same path must also lead to a $\Diamond$-label, since no other label can satisfy the $\sqsubseteq$-requirement. Hence, the path from y to y in A_1 translates to a path from $h(y)$ to $h(y)$ in A_2. It follows that $z = h(y)$.

Similarly, injectivity follows. If for some x' we have $z = h(x')$, then the cycle from z to z in A_2 must correspond to a cycle in A_1, from which it follows that $x = x'$.

Since all states in a canonical automaton can be reached from the initial state, this construction will terminate with a total function.

To proceed, we observe that the existence of *any* injective function from states of A_1 to states of A_2 assures that there are no more states in A_1 than in A_2. Since any label in A_1 must be $\leq$ than some label in A_2, and we know that $\leq$ has finite intervals, then any A_1 must be built out of a bounded number of states and a finite set of labels. For simple combinatorial reasons, there can only be finitely many such automata.

Since different L-trees yield different canonical automata, the result follows. $\Box$

In particular, this result means that any class can only have finitely many superclasses.

Corollary 5.11: For any two L-trees T_1, T_2 the closed interval $\{S \mid T_1 \lhd S \lhd T_2\}$ is finite.
Proof: This is just a subset of the finite interval in proposition 5.10. $\Box$

Next, we can show that our generalized notion of subclassing does not allow any temporal cycles. Since in our framework

$$T_1 \text{ HAS-A } T_2 \;\; \textit{iff} \;\; \exists \alpha : \; T_1 \downarrow \alpha = T_2$$

and

$$T_1 \text{ IS-A } T_2 \;\; \textit{iff} \;\; T_2 \lhd T_1 \;\wedge\; T_2 \neq T_1$$

then, to eliminate temporal cycles of the form T HAS-A S IS-A T, we must show that no tree can be strictly $\lhd$-less than one of its subtrees. Longer cycles are handled by transitivity and essentially the same argument.

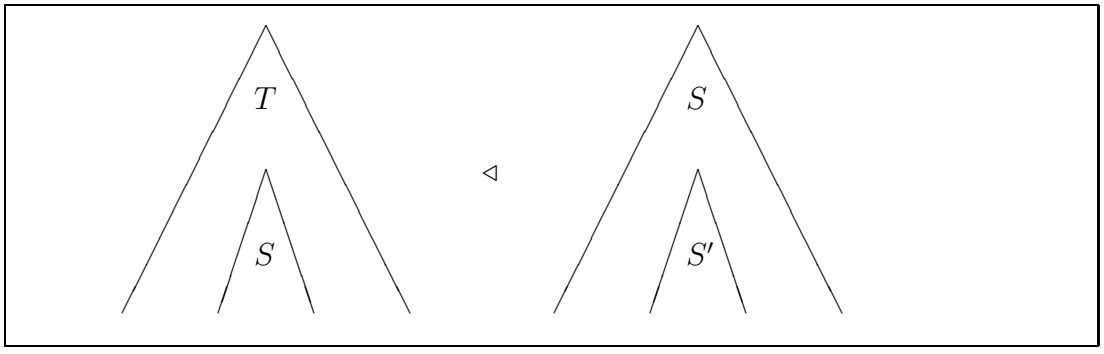

Figure 10: A temporal cycle.

Theorem 5.12: Let T by an L-tree and let S be a subtree of T. If $T \lhd S$, then

$T = S$.
Proof: Suppose $T \lhd S$ and $T \neq S$. Choose α so that $T \downarrow \alpha$ is a maximal occurrence of S in T. Let $S' = S \downarrow \alpha$. Then $S \lhd S'$, as illustrated in figure 10. We will now show that $S \neq S'$. Suppose $S = S'$. There are two cases. First, if the generator of S has a $\Diamond$-label in position α, then also T has a $\Diamond$-label in position α. This implies $T = S$, a contradiction. Second, if the generator of S does not have a $\Diamond$-label in position α, then we can find a maximal proper occurrence of S' in S. Let this occurrence be in position β, where β is a proper prefix of α. Thus, the generator of S has a $\Diamond$-label in position β, so also T has a $\Diamond$-label in position β. Suppose $\alpha = \beta\gamma$. Clearly, $T \downarrow \gamma = S$, contradicting that $T \downarrow \alpha$ is a maximal occurrence of S in T. We have thus proved that $S \neq S'$. By iterating the above construction we obtain a strictly $\lhd$-increasing chain

$$T \lhd T \downarrow \alpha \lhd T \downarrow \alpha^2 \lhd \downarrow \alpha^3 \lhd \cdots \lhd T \downarrow \alpha^i \lhd \cdots$$

This means that T has infinitely many different subtrees, contradicting its being an L-tree. $\Box$

A final property can be phrased as the slogan *subclassing preserves subtyping*. Intuitively, this means that subtype relationships will be preserved in subclasses.

Proposition 5.13: A type expression in a class T consists of, say, n classes located as the subtrees at addresses $\alpha_1, \alpha_2, \ldots, \alpha_n$; that is, the expression denotes the set

$$A = \{T \downarrow \alpha_1, T \downarrow \alpha_2, \ldots, T \downarrow \alpha_n\}$$

Suppose also we have another type expression denoting the set

$$B = \{T \downarrow \beta_1, T \downarrow \beta_2, \ldots, T \downarrow \beta_m\}$$

and that the inclusion $A \subseteq B$ holds. Let S be any subclass of T. We then have two corresponding sets

$$A' = \{S \downarrow \alpha_1, S \downarrow \alpha_2, \ldots, S \downarrow \alpha_n\} \quad B' = \{S \downarrow \beta_1, S \downarrow \beta_2, \ldots, S \downarrow \beta_m\}$$

and we are guaranteed that the inclusion $A' \subseteq B'$ will also hold.
Proof: This follows immediately from the stability requirement on $\lhd$. $\Box$

5.4 Examples

To illustrate the concepts introduced in this section we continue the example from subsection 3.4.

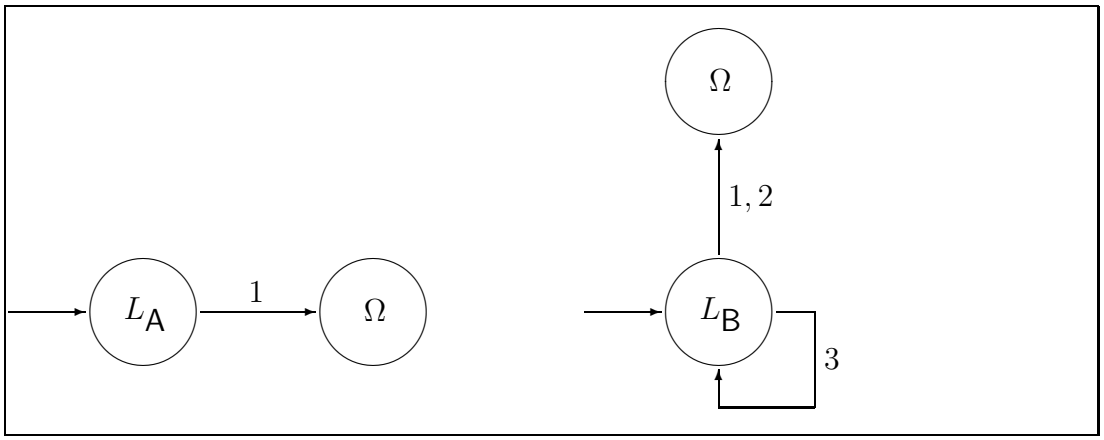

Figure 11: Example automata.

```
class C
    var h: B
    var t: selfClass
end C

class D inherits C
    var z: Object
end D
```

Figure 12: Example classes.

The automata corresponding to the classes A and B are shown in figure 11. We can also observe that $\textsc{tree}(\mathsf{A}) \sqsubseteq \textsc{tree}(\mathsf{B})$.

We next program two new classes C and D shown in figure 12. As before, we define

$$L_{\mathsf{C}} = \textbf{var } \mathsf{h}\text{: } \bullet \textbf{ var } \mathsf{t}\text{: } \bullet$$
$$L_{\mathsf{D}} = \textbf{var } \mathsf{z}\text{: } \bullet$$

The corresponding trees, shown in figure 13, are defined by the equations

$$\textsc{tree}(\mathsf{C}) = L_{\mathsf{C}}(\textsc{tree}(\mathsf{B}), \textsc{tree}(\mathsf{C}))$$
$$\textsc{tree}(\mathsf{D}) = L_{\mathsf{C}}L_{\mathsf{D}}(\textsc{tree}(\mathsf{B}), \textsc{tree}(\mathsf{D}), \textsc{tree}(\mathsf{Object}))$$

Let us show that $\textsc{tree}(\mathsf{C}) \lhd \textsc{tree}(\mathsf{D})$. We have three different situations where a generator in $\textsc{tree}(\mathsf{C})$ must be $\preceq$ than a similar generator in $\textsc{tree}(\mathsf{D})$. Examples of all three situations are shown in figure 14. A tree that is $\preceq$ than $\textsc{tree}(\mathsf{D})$ but not $\lhd$ is shown in figure 15; it is not recursive, while $\textsc{tree}(\mathsf{D})$ is.

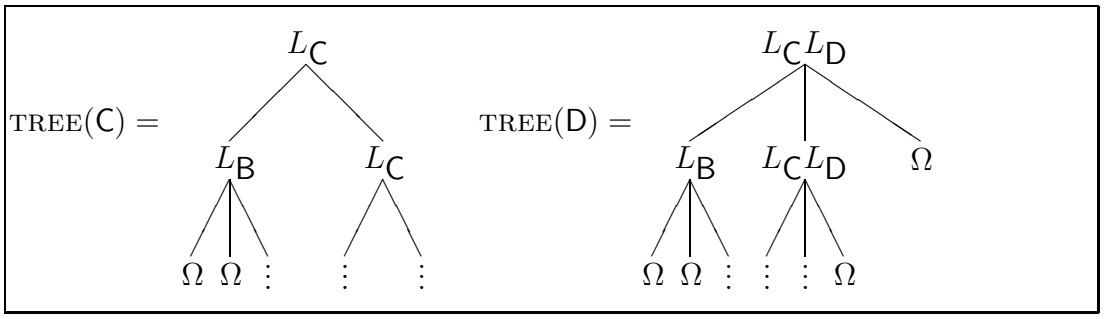

Figure 13: Example trees.

Figure 14: Relating generators.

6 The Orthogonality Result

Inheritance is a programming mechanism which can realize only part of $\lhd$; more precisely, it captures a suborder.

6.1 Two Suborders

Definition 6.1: The partial order $T_1 \lhd_I T_2$ holds exactly when

- $T_1 \lhd T_2 \ \wedge \ \forall \alpha \in T_1 : \ T_1 \downarrow \alpha \neq T_1 \Rightarrow T_1[\alpha] = T_2[\alpha]$

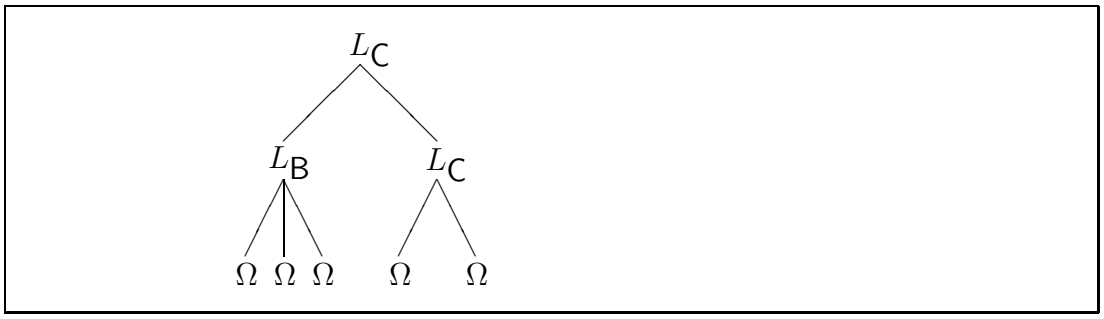

Figure 15: A non-recursive tree.

This states that only the root label—and its recursive occurrences— may change. □

The $\lhd_I$-part of $\lhd$ *is* just inheritance.

Proposition 6.2: If C_1 is inherited by C_2 in any program, then $\textsc{tree}(\mathsf{C}_1) \lhd_I \textsc{tree}(\mathsf{C}_2)$. Conversely, if $T_1 \lhd_I T_2$ then any C_1 such that $\textsc{tree}(\mathsf{C}_1) = T_1$ can be modified by inheritance to yield a C_2 such that $\textsc{tree}(\mathsf{C}_2) = T_2$.
Proof: Consider the isomorphism between L-trees and minimal equation systems. It is quite easy to see that $\lhd_I$ in this framework exactly captures the constructions performed by the expansion algorithm. □

The remaining part of $\lhd$ can be characterized in a satisfying manner: as an orthogonal complement of $\lhd_I$, in the following sense.

Definition 6.3: Let P be a partial order on a set S. We use the notation $\triangle(S)$ for the diagonal $\{(s,s) \mid s \in S\}$, and the notation A^* for the reflexive, transitive closure of a relation A. We write $Q \perp_P R$, if Q and R are partial orders such that $Q \cap R = \triangle(S)$ and $(Q \cup R)^* = P$. We call Q, R an *orthogonal basis* for P when

- $Q \perp_P R$
- $Q' \perp_P R \;\Rightarrow\; Q \subseteq Q'$
- $Q \perp_P R' \;\Rightarrow\; R \subseteq R'$

This generalizes the notion of *basis* in [26]. □

For example, if $(S_1, \leq_1)$ and $(S_2, \leq_2)$ are partial orders, then $\leq_1 \times \triangle(S_2)$ and $\triangle(S_1) \times \leq_2$ form an orthogonal basis for $\leq_1 \times \leq_2$.

The remaining part of $\lhd$ can be captured by the following suborder.

Definition 6.4: The partial order $T_1 \lhd_S T_2$ holds exactly when

- $T_1 \lhd T_2 \ \wedge \ T_1[\lambda] = T_2[\lambda]$

This states that the root label must remain unchanged. □

6.2 Orthogonality

We can now show that $\lhd_I$,$\lhd_S$ is an orthogonal basis for $\lhd$. This result is important, since it allows us to simply search for a programming mechanism that relates to $\lhd_S$ in the same fashion that inheritance relates to $\lhd_I$; the less appealing choice was to find a mechanism directly for the awkward set difference of $\lhd$ and $\lhd_I$. Furthermore, when we have such a $\lhd_S$-mechanism, then it is orthogonal to inheritance in a formal sense. The next chapter discloses that $\lhd_S$ yields a form of *genericity*. We have thus shown that inheritance and genericity are independent, orthogonal components of generalized subclassing.

To prove the result we need a series of lemmas.

Lemma 6.5: The relations $\lhd_I, \lhd_S$ are both partial orders, and $\lhd_I \cap \lhd_S = \triangle(\mathcal{U})$.
Proof: Clearly, $\lhd_S$ is a partial order. The extra condition on $T_1 \lhd_I T_2$ simply means that for every $\alpha \in \textsc{gen}(T_1)$ we have $\textsc{gen}(T_1)[\alpha] = \textsc{gen}(T_2)[\alpha]$, except for the root labels which are $\leq$-related. Hence, $\lhd_I$ is a partial order. If also $T_1 \lhd_S T_2$ then *all* labels must be equal, so the generators, and the trees, are equal. □

Lemma 6.6: Whenever $T_1 \lhd T_2$ then there is a unique $A \in \mathcal{U}$ such that $T_1 \lhd_S A \lhd_I T_2$.
Proof: Suppose $T_1 \lhd T_2$. Then $\textsc{gen}(T_1) \preceq \textsc{gen}(T_2)$. Let L_1 be the root label of $\textsc{gen}(T_1)$. Then the root label of $\textsc{gen}(T_2)$ must look like $L_1 L_2$. Let $\textsc{gen}(A)$ be obtained from $\textsc{gen}(T_2)$ by removing the L_2-part of the root label and the subtrees that correspond to its gaps. Since subtrees with the same address in $\lhd$-related trees also will be $\lhd$-related, it follows that $T_1 \lhd A$. Since $T_1, T_2 \in \mathcal{U}$, then clearly $A \in \mathcal{U}$. But since they both have root label L_1, we also have $T_1 \lhd_S A$. It is trivially the case that $A \lhd_I T_2$, so we have shown that $T_1 \lhd_S A \lhd_I T_2$.

For the uniqueness of A, suppose we also have $T_1 \lhd_S B \lhd_I T_2$. Then for every $\alpha \in \text{GEN}(T_2)$ we have $\text{GEN}(T_2)[\alpha] = \text{GEN}(A)[\alpha] = \text{GEN}(B)[\alpha]$, except for the root labels; but we also have $T_1[\lambda] = A[\lambda] = B[\lambda]$, so $A = B$. $\square$

Lemma 6.7: $(\lhd_I \cup \lhd_S)^* = \lhd$
Proof: Immediate from lemma 6.6. $\square$

Lemma 6.8: No partial order $\lhd_M$ which is a proper subset of $\lhd_S$ satisfies $(\lhd_I \cup \lhd_M)^* = \lhd$. Also, no partial order $\lhd_N$ which is a proper subset of $\lhd_I$ satisfies $(\lhd_N \cup \lhd_S)^* = \lhd$.
Proof: Suppose we have such a $\lhd_M$. Choose $(x, y) \in \lhd_S \backslash \lhd_M$. Then $x[\lambda] = y[\lambda]$, so no $\lhd_I \backslash \triangle(\mathcal{U})$ steps can take place on a path from x to y. Hence, $(x, y) \in \lhd_M^* = \lhd_M$, which is a contradiction. The other half of the result is proved similarly. $\square$

Lemma 6.9: Let P be a partial order, where all closed intervals are finite. Whenever $P_1, P_2 \subseteq P$ and $P_1^* = P_2^* = P$, then $(P_1 \cap P_2)^* = P$.
Proof: Clearly, $(P_1 \cap P_2)^* \subseteq P$. For the opposite inclusion, suppose $(x, y) \in P$. The proof is by induction in the size of the open interval over P from x to y. If the interval is empty, then either $(x, y) \in (P_1 \cap P_2)^0$, or $(x, y) \in (P_1 \cap P_2)$. Now, suppose the interval contains $n+1$ elements. Choose z in it. Then both the open interval from x to z and that from z to y contain at most n elements. Hence, by the induction hypothesis, $(x, z), (z, y) \in (P_1 \cap P_2)^*$. By transitivity of $(P_1 \cap P_2)^*$ we conclude $(x, y) \in (P_1 \cap P_2)^*$. $\square$

Lemma 6.10: If $\lhd_M \perp_\lhd \lhd_S$ then $\lhd_I \subseteq \lhd_M$. Also, if $\lhd_I \perp_\lhd \lhd_N$ then $\lhd_S \subseteq \lhd_N$.
Proof: Suppose $\lhd_M \perp_\lhd \lhd_S$. By corollary 5.11, all closed $\lhd$-intervals of $\mathcal{U}$ are finite, so by lemmas 6.7 and 6.9, $\lhd = ((\lhd_I \cup \lhd_S) \cap (\lhd_M \cup \lhd_S))^* = ((\lhd_I \cap \lhd_M) \cup \lhd_S)^*$. By lemma 6.8, $\lhd_I \cap \lhd_M$ cannot be a proper subset of $\lhd_I$. Hence, $\lhd_I \cap \lhd_M = \lhd_I$, so $\lhd_I \subseteq \lhd_M$. The other half of the lemma is proved similarly. $\square$

Theorem 6.11: $\lhd_I, \lhd_S$ is an orthogonal basis for $\lhd$.
Proof: Combine lemmas 6.5, 6.7, and 6.10. $\square$

The significance of this result is that a programming mechanism realizing $\lhd_S$, together with inheritance which realizes $\lhd_I$, allow the programming of all subclasses. Furthermore, two such programming mechanisms would be completely non-redundant; neither could emulate the other. The situation is illustrated in

figure 16, which shows how all subclasses can be reached in axis-parallel moves.

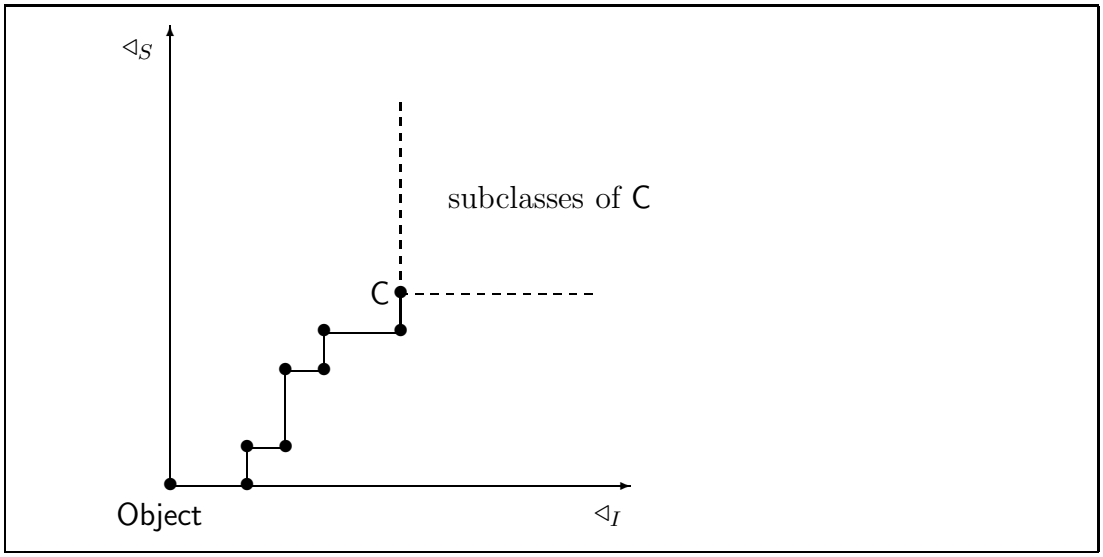

Figure 16: Orthogonal suborders.

7 Class Substitution

The suborder $\lhd_S$ requires that the root label cannot change. In terms of classes, this means that only type information may change, and not the implementation itself. This is precisely what intuitively characterizes *genericity*. This section introduces a programming mechanism that corresponds to $\lhd_S$. It is called *class substitution* and implements a general form of genericity.

7.1 Syntax

The syntax for class substitution is as follows. If C, A_i, and B_i are classes, then

$$\mathsf{C}[\mathsf{A}_1, \ldots, \mathsf{A}_n \leftarrow \mathsf{B}_1, \ldots, \mathsf{B}_n]$$

is a *class substitution* which specifies a class D such that $\mathsf{C} \lhd_S \mathsf{D}$ and all occurrences of A_i are substituted by B_i. If such a class does not exist, then the specification is statically incorrect.

7.2 Semantics

In this section we define precisely what class substitution means, and we show that it exactly realizes $\lhd_S$.

Input:	A substitution specification: $\mathsf{C}[\mathsf{A}_1, \ldots, \mathsf{A}_n \leftarrow \mathsf{B}_1, \ldots, \mathsf{B}_n]$
Output:	Either *fail* or a resulting class: D
Algorithm:	$M \leftarrow \{(\mathsf{A}_i \downarrow \alpha, \mathsf{B}_i \downarrow \alpha) \mid 1 \leq i \leq n,\ \alpha \in \mathsf{A}_i,\ \alpha \in \mathsf{B}_i\}$ if M is not a $\lhd$-increasing, partial function then *fail* apply M to the maximal subtrees of C in $dom(M)$ yielding D if not $\mathsf{D} \in \mathcal{U}$ and $\mathsf{C} \lhd_S \mathsf{D}$ then *fail*

Figure 17: Expanding substitutions.

The algorithm to expand a substitution specification is summarized in figure 17. Intuitively, the relation M collects all the individual substitutions that must be performed. The requirement that M is $\lhd$-increasing is necessary in order for D to be a subclass of C. The requirement that M is a function is necessary to avoid inconsistent substitutions. Note that the maximal subtrees of C that belong to the domain of M is a uniquely determined set of disjoint subtrees. Note also that failed substitutions can be determined on compile-time.

Proposition 7.1: If $\mathsf{C} \lhd_S \mathsf{D}$ then

$$\mathsf{D} = \mathsf{C}[\mathsf{A}_1, \ldots, \mathsf{A}_n \leftarrow \mathsf{B}_1, \ldots, \mathsf{B}_n]$$

for some $\mathsf{A}_i, \mathsf{B}_i$.
Proof: Clearly, $\mathsf{D} = \mathsf{C}[\mathsf{C} \leftarrow \mathsf{D}]$. □

Hence, all $\lhd_S$-related subclasses can be expressed in this manner. Note though that the specification given in the proof of proposition 7.1 is rather useless for practical programming: it corresponds to writing class D from scratch. There are often many different specifications of the same class substitution, however, and in the following section we will see how easy it is to select a convenient one.

It is also for pragmatic reasons that we allow multiple, *simultaneous* substitutions. With this mechanism conflicting substitutions will cause compile-time errors; in contrast, a sequence of individual substitutions will always succeed but may yield an unexpected result. For example, consider $\mathsf{C}[\mathsf{Object},\mathsf{Object} \leftarrow \mathsf{Integer},\mathsf{Boolean}]$. Clearly, this specification should lead to a compile-time error. But if we first carry out $\mathsf{C}[\mathsf{Object} \leftarrow \mathsf{Integer}]$ and then $\mathsf{C}[\mathsf{Object} \leftarrow \mathsf{Boolean}]$, then both succeed because the second will have no effect.

With this expansion algorithm at hand, we can now consider expanding programs with more than one substitution specification. Using a well-formedness criterion similar to the one presented in Section 4, the substitution specifications can be

sorted and expanded one by one. For a detailed explanation and examples of this, see [46].

7.3 Pragmatics

The fact that class substitution realizes $\lhd_S$ is not sufficient to ensure that class substitution is a useful and pleasant programming mechanism. Only pragmatic arguments can really justify such a claim. In this section we attempt to give such arguments by showing some example programs which use class substitution, and by comparing class substitution with parameterized classes.

In figure 18 is shown a subclass hierarchy as it can be programmed using inheritance and class substitution. An arrow is labeled by "I" when the subclass is obtained by inheritance, and "S" if by class substitution. The detailed code for the classes will be given subsequently. We assume that the classes Boolean, Integer, and Array has been programmed already, and that Array instances respond to messages as specified in figure 19.

In the class Stack, the element type is Object, see figure 19. The classes Booleanstack and Integerstack are class substitutions of Stack. For example, Booleanstack is the class obtained from Stack by substituting *all* occurrences of Object by Boolean, including those in Array. Thus, Stack acts like a parameterized class but is just a class, not a second-order entity. This enables *gradual* generic-instantiations, as demonstrated below.

The use of parameterized classes is the traditional approach to *genericity* [38, 39, 50, 34, 49, 42]. Similar constructs are found in conventional procedural languages, for example Ada generic packages [24], and parameterized Clu clusters [35]. A parameterized class is a second-order entity which is generically-instantiated to specific classes when actual type parameters are supplied. Generic-instantiation of parameterized classes is less flexible than inheritance, since any class can be inherited but is not in itself parameterized. In other words, code reuse with parameterized classes requires planning; code reuse with inheritance does not.

In general, we will say that a genericity mechanism is a construct which allows the substitution of types in a class. Thus, class substitution is also a genericity mechanism, and in contrast to parameterized classes it is the orthogonal complement of inheritance. This indicates that class substitution gives more expressive power to an object-oriented language than parameterized classes, and indeed Meyer [38] argued that parameterized classes can be simulated by inheritance.

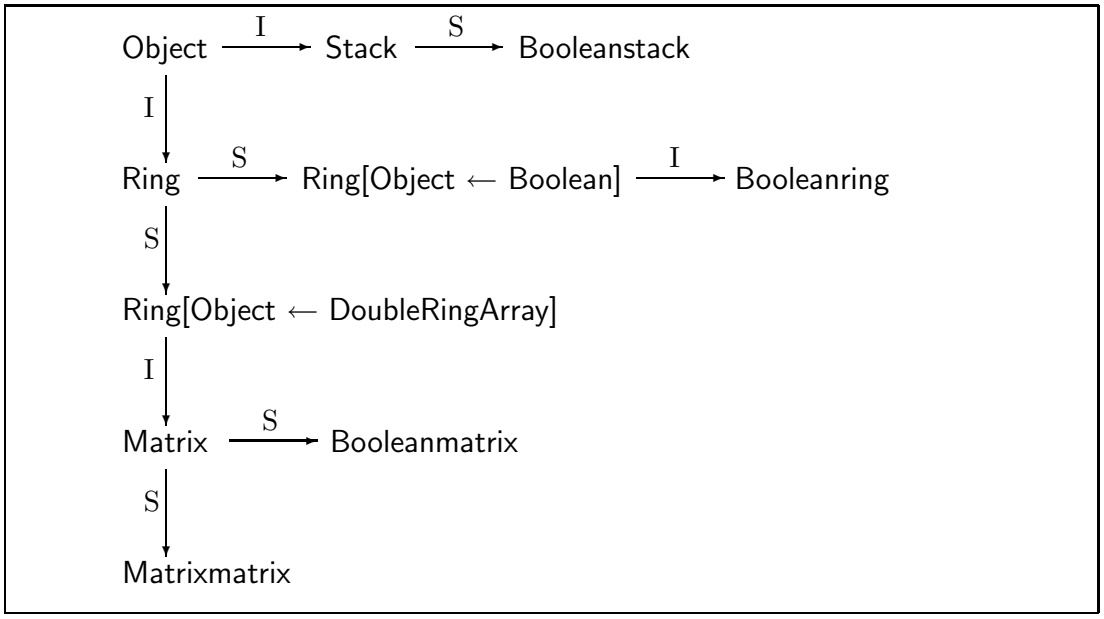

Figure 18: Programming with class substitution.

```
class Array
    method at(i: Integer) returns Object ...
    method atput(i: Integer ; x: Object) returns selfClass ...
    method initialize(size: Integer) returns selfClass ...
    method arraysize returns Integer ...
    ...
end Array

class Stack
    var space: Array
    var index: Integer
    method push(x: Object) returns selfClass
        index := index succ ; space.atput(index,x) ; self
    method top returns Object
        space.at(index)
    method pop returns selfClass
        index := index pred ; self
    method initialize(size: Integer) returns selfClass
        space := (Array new).initialize(size) ; index := 0 ; self
    ...
end Stack

class Booleanstack is Stack[Object ← Boolean]
class Integerstack is Stack[Object ← Integer]
```

Figure 19: Stack classes.

```
class Ring
    var value: Object
    method plus(other: selfClass) returns selfClass
        self
    method zero returns selfClass
        self
    method getvalue returns Object
        value
    ...
end Ring

class BooleanRing inherits Ring[Object ← Boolean]
    method plus(other: selfClass) returns selfClass
        value := value.or(other.getvalue) ; self
    method zero returns selfClass
        value := false ; self
    ...
end BooleanRing

class DoubleArray is Array[Object ← Array]
class DoubleRingArray is DoubleArray[Object ← Ring]

class Matrix inherits Ring[Object ← DoubleRingArray]
   var i,j: Integer
   var r: Array[Object ← Ring]
    method plus(other: selfClass) returns selfClass
        for i := 1 to value.arraysize do
            r := value.at(i) ;
            for j:=1 to r.arraysize do
                r := r.atput(j, r.at(j).plus(other.at(i).at(j)))
            od ;
            value.atput(i,r)
        od ;
        self
    ...
end Matrix

class Booleanmatrix is Matrix[Ring ← Booleanring]
class Matrixmatrix is Matrix[Ring ← Matrix]
```

Figure 20: Ring and matrix classes.

Note that class substitution is *not* textual substitution; in general, textual substitution will not yield a statically correct subclass. For example, if we try to obtain Booleanstack by textually substituting occurrences of Object by Boolean, then among others the expression space.atput(index,x) (in push) becomes statically incorrect; it will have an Boolean instance where an Object instance is required. For further examples, see [43].

Another drawback of parameterized classes is that they cannot be gradually generically-instantiated. This makes it awkward to, for example, declare a class Ring, then specialize it to a class Matrix, and finally specialize Matrix to a class Booleanmatrix. In the following we show how it can be done using inheritance and class substitution. The *history* of a class developed with inheritance and substitution can be thought of as an element of the language $(\mathrm{I} + \mathrm{S})^*$; in comparison, parameterized classes restrict the possible histories to $\mathrm{S.I}^*$.

Consider the ring classes in figure 20. The class Booleanring inherits a class substitution of class Ring; thus, Booleanring is a subclass of Ring. This illustrates how class substitution and inheritance complement each other: first Object is substituted by Boolean; then the inherited procedures are implemented appropriately.

This is further illustrated by the matrix classes, see figure 20. Again, the class Matrix is obtained through a class substitution followed by an application of inheritance. We take the liberty of using a for-statement, even though is has not been included in the syntax. Class Matrixmatrix is obtained through class substitution alone.

It seems that class substitution could solve the problems in the EIFFEL type system that were reported by Cook [18], since attributes cannot be redeclared in *isolation* in subclasses, there are no *asymmetries* as with declaration by association, and generic-instantiation can be expressed as subclassing.

8 Separate Compilation and Infinite Types

One common aspect of many existing object-oriented languages has so far not been captured by our framework.

We have restricted our types to be *finite* sets. We argued that this was quite adequate, since in a given program any predicate could only be satisfied by finitely many classes.

With this type system we have developed a general notion of subclassing, under which any subclass can be implemented as an extension of its superclass. This is the fundamental idea of *code reuse* in object-oriented programming.

The concept of *separate compilation*, however, introduces a different level of code

reuse, which does not fit as smoothly into our framework. When a class is separately compiled, then predicates cannot be expanded into finite sets, since only a part of the program is known at the time of compilation.

The traditional solution is to introduce a limited form of infinite sets; in particular, *cones* of the form

$$\uparrow T = \{S \in \mathcal{U} \mid T \lhd S\}$$

are employed. It is possible to generalize slightly: finite unions of cones and singletons can be used. The important restriction is that the sets can be finitely represented, and that membership and mutual inclusions are decidable. For cones, membership is just subclassing, and inclusion coincides with reverse subclassing, i.e.

$$\uparrow S \subseteq \uparrow T \quad \textit{iff} \quad T \lhd S$$

A perfect match is not possible, but the types in a language such as BETA correspond closely to either singletons or cones [36].

Recall the important property of our framework that can be stated as the slogan: *subclassing preserves subtyping*. In the presence of cones, the picture changes dramatically. There exist classes $T \lhd S$ such that for some α, β we have

$$\uparrow(T \downarrow \alpha) \subseteq \uparrow(T \downarrow \beta) \quad \textit{but} \quad \uparrow(S \downarrow \alpha) \not\subseteq \uparrow(S \downarrow \beta)$$

This unfortunately means that subclasses cannot be guaranteed to remain statically correct. A simple example is shown in figure 21, in which S is a statically *incorrect* subclass of the statically *correct* class T.

```
class T
    var x: ↑Object
    var y: ↑Integer
    method Assign returns selfClass
        x := y ; self
end T

class S
    var x: ↑Boolean
    var y: ↑Integer
    method Assign returns selfClass
        x := y ; self
end S
```

Figure 21: A statically incorrect subclass.

There seem to be three possible solutions.

- *restrict subclassing to preserve* $\subseteq$: unfortunately, only trivial subclasses can be allowed.
- *restrict subtyping to be preserved by* $\lhd$: unfortunately, only trivial subtypes can be allowed.
- *find a useful compromise between both subclassing and subtyping*: unfortunately, no such compromise seems to be forthcoming.

The situation does not look hopeful. The choice made by real-life languages is to keep both $\subseteq$ and $\lhd$, which leads to an statically unsound type system. The reactions to such a predicament fall on a spectrum ranging from C++, in which these loop-holes are simply ignored, to BETA, in which the necessary run-time type-checks are inserted into the code of the superclass, yielding a dynamically sound type system.

We consider a more satisfactory solution to the problem of separate compilation to be an extremely hard challenge.

9 Conclusion

Our type system for object-oriented languages is conceptually simple and it ensures that programs are readable, reliable, and more efficient. Our subclassing order can be implemented straightforwardly and it contains inheritance and class substitution as an orthogonal basis.

It is too preliminary to judge the pragmatics of class substitution, but several examples indicate that it may be a useful alternative to parameterized classes. Future work includes implementation and experimentation with the mechanism.

The expansion algorithm for transforming a program that uses inheritance into one which does not may be of interest in itself. It can be useful for programmers who want to use inheritance but are required to implement their programs in a language which does not support it.

Further explanations of the expansion algorithms in this paper are given in [46].

Acknowledgement. The authors thank Ole Lehrmann Madsen, Peter Mosses, and Flemming Nielson for helpful comments on drafts of the paper.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company,

1974.

[2] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Also in Proc. POPL'91.

[3] Grady Booch. *Object-Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, 1991.

[4] Alan H. Borning and Daniel H. H. Ingalls. A type declaration and inference system for Smalltalk. In *Ninth Symposium on Principles of Programming Languages*, pages 133–141, 1982.

[5] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. OOPSLA/ECOOP'90, ACM SIGPLAN Fifth Annual Conference on Object-Oriented Programming Systems, Languages and Applications; European Conference on Object-Oriented Programming*, pages 303–311, 1990.

[6] Ronald J. Brachman. "I lied about the trees" or, defaults and definitions in knowledge representation. *The AI Magazine*, 6(3):80–93, 1985.

[7] Kim B. Bruce. The equivalence of two semantic definitions for inheritance in object-oriented languages. In *Proc. Mathmatical Foundations of Programming Semantics*, pages 102–124. Springer-Verlag (*LNCS* 598), 1992.

[8] Peter S. Canning, William R. Cook, Walter L. Hill, John Mitchell, and Walter G. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[9] Peter S. Canning, William R. Cook, Walter L. Hill, and Walter G. Olthoff. Interfaces for strongly-typed object-oriented programming. In *Proc. OOPSLA'89, Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 457–467. ACM, 1989.

[10] Luca Cardelli. A semantics of multiple inheritance. In Gilles Kahn, David MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types*, pages 51–68. Springer-Verlag (*LNCS* 173), 1984.

[11] Luca Cardelli. Typeful programming. Technical Report No. 45, Digital Equipment Corporation, Systems Research Center, 1989.

[12] Luca Cardelli and John C. Mitchell. Operations on records. In *Proc. Mathmatical Foundations of Programming Semantics*, pages 22–52. Springer-Verlag (*LNCS* 442), 1989.

[13] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[14] Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Prentice Hall, 1991.

[15] William Cook, Walter Hill, and Peter Canning. Inheritance is not subtyping. In *Seventeenth Symposium on Principles of Programming Languages*, pages 125–135, 1990.

[16] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, 1994. Also in Proc. OOPSLA'89, ACM SIGPLAN Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications, pages 433–443, New Orleans, Louisiana, October 1989.

[17] William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[18] William R. Cook. A proposal for making Eiffel type-safe. *The Computer Journal*, 32(4):305–311, 1989.

[19] Bruno Courcelle. Infinite trees in normal form and recursive equations having a unique solution. *Mathematical Systems Theory*, 13:131–180, 1979.

[20] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(1):95–169, 1983.

[21] Brad J. Cox. *Object Oriented Programming, an Evolutionary Approach*. Addison-Wesley Publishing Company, 1986.

[22] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. Simula 67 common base language. Technical report, Norwegian Computing Center, Oslo, Norway, 1968.

[23] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[24] Jean D. Ichbiah et al. *Reference Manual for the Ada Programming Language*. US DoD, July 1982.

[25] Adele Goldberg and David Robson. *Smalltalk-80—The Language and its Implementation*. Addison-Wesley, 1983.

[26] George Grätzer. *General Lattice Theory*. Birkhäuser, 1978.

[27] Justin O. Graver and Ralph E. Johnson. A type system for Smalltalk. In *Seventeenth Symposium on Principles of Programming Languages*, pages 136–150, 1990.

[28] Justin Owen Graver. *Type-Checking and Type-Inference for Object-Oriented Programming Languages*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1989. UIUCD-R-89-1539.

[29] Ralph E. Johnson. Type-checking Smalltalk. In *Proc. OOPSLA'86, Object-Oriented Programming Systems, Languages and Applications*, pages 315–321. Sigplan Notices, 21(11), November 1986.

[30] Ralph E. Johnson and Brian Foote. Designing reuable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[31] Samuel Kamin. Inheritance in Smalltalk–80: A denotational definition. In *Fifteenth Symposium on Principles of Programming Languages*, pages 80–87, 1988.

[32] Bent B. Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The BETA programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, 1987.

[33] Bent B. Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Classification of actions or inheritance also for methods. In *Proc. ECOOP'87, European Conference on Object-Oriented Programming*, pages 98–107. Springer-Verlag (*LNCS* 276), 1987.

[34] Karl J. Lieberherr and Arthur J. Riel. Contributions to teaching object-oriented design and programming. In *Proc. OOPSLA'89, Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 11–22. ACM, 1989.

[35] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Scaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[36] Ole Lehrmann Madsen, Boris Magnusson, and Birger Møller-Pedersen. Strong typing of object-oriented languages revisited. In *Proc. OOPSLA/ECOOP'90, ACM SIGPLAN Fifth Annual Conference on Object-Oriented Programming Systems, Languages and Applications; European Conference on Object-Oriented Programming*, pages 140–150, 1990.

[37] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proc. OOPSLA'89, Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 397–406. ACM, 1989.

[38] Bertrand Meyer. Genericity versus inheritance. In *Proc. OOPSLA'86, Object-Oriented Programming Systems, Languages and Applications*, pages 391–405. Sigplan Notices, 21(11), November 1986.

[39] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[40] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[41] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Seventeenth Symposium on Principles of Programming Languages*, pages 109–124, 1990.

[42] Atsushi Ohori and Peter Buneman. Static type inference for parametric classes. In *Proc. OOPSLA'89, Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 445–456. ACM, 1989.

[43] Jens Palsberg and Michael I. Schwartzbach. Type substitution for object-oriented programming. In *Proc. OOPSLA/ECOOP'90, ACM SIGPLAN Fifth Annual Conference on Object-Oriented Programming Systems, Languages and Applications; European Conference on Object-Oriented Programming*, pages 151–160, Ottawa, Canada, October 1990.

[44] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proc. OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161, Phoenix, Arizona, October 1991.

[45] Jens Palsberg and Michael I. Schwartzbach. What is type-safe code reuse? In *Proc. ECOOP'91, Fifth European Conference on Object-Oriented Programming*, pages 325–341. Springer-Verlag (*LNCS* 512), Geneva, Switzerland, July 1991.

[46] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.

[47] Claus H. Pedersen. Extending ordinary inheritance schemes to include generalization. In *Proc. OOPSLA'89, ACM SIGPLAN Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 407–418, 1989.

[48] Uday S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 289–297, 1988.

[49] David Sandberg. An alternative to subclassing. In *Proc. OOPSLA'86, Object-Oriented Programming Systems, Languages and Applications*, pages 424–428. Sigplan Notices, 21(11), November 1986.

[50] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Proc. OOPSLA'86, Object-Oriented Programming Systems, Languages and Applications*, pages 9–16. Sigplan Notices, 21(11), November 1986.

[51] Michael I. Schwartzbach. Static correctness of hierarchical procedures. In *Proc. International Colloquium on Automata, Languages, and Programming 1990*, pages 32–45. Springer-Verlag (*LNCS* 443), 1990.

[52] Michael I. Schwartzbach. Type inference with inequalities. In *Proc. TAPSOFT'91*, pages 441–455. Springer-Verlag (*LNCS* 493), 1991.

[53] Alan Snyder. Inheritance and the development of encapsulated software components. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 165–188. MIT Press, 1987.

[54] Bjarne Stroustrup. *The* C++ *Programming Language*. Addison-Wesley, 1986.

[55] David S. Touretzky. *The Mathematics of Inheritance Systems*. Morgan Kaufmann Publishers, 1986.

[56] Peter Wegner. Dimensions of object-based language design. In *Proc. OOPSLA'87, Object-Oriented Programming Systems, Languages and Applications*, pages 168–182, 1987.

[57] Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *Proc. ECOOP'88, European Conference on Object-Oriented Programming*, pages 55–77. Springer-Verlag (*LNCS* 322), 1988.

[58] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, New York, 1985.

Contents