A Unified Type System for Object-Oriented Programming

Jens Palsberg
palsberg@daimi.au.dk

Michael I Schwartzback
mis@daimi.au.dk

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

June 1993

Abstract

We present a new type system for object-oriented languages with assignments. Types are sets of classes, subtyping is set inclusion, and genericity is class substitution. The type system enables separate compilation, and unifies, generalizes, and simplifies the type systems underlying SIMULA/BETA, C++, and EIFFEL, and Typed Smalltalk, and one with type substitution proposed by Palsberg and Schwartzback. Classes and types are both modeled as node-labeled, ordered regular trees; this allows an efficient type-checking algorithm.

Keywords: polymorphism, assignments, subtyping, subclassing.

1 Introduction

Many object-oriented languages feature inheritance, variables, and assignments. Major examples are SMALLTALK [11], SIMULA [10], BETA [14], C++ [22], and EIFFEL [17]. In order to guarantee that the run-time error Message-not-understood will never occur, a number of type systems for such languages have been proposed. These type systems differ markedly when comparing their notion of *type*, *polymorphism*, and *genericity*.

This paper presents a new type system where types are sets of classes, subtyping is set inclusion, and genericity is class substitution. It avoids type variables and second-order entities, and enables separate compilation. It also provides a clear distinction between types and classes, and between subtyping and subclassing. A type is a *specification*; a class describes an *implementation*. Subtyping supports polymorphic *applications*, and subclassing supports reuse of class *definitions* [21]. Our type system thus unifies, generalizes, and simplifies the type systems underlying SIMULA/BETA, C++, EIFFEL(the SIMULA school, henceforth denoted SS), and Typed Smalltalk [12, 13] (henceforth denoted TS), and one with type substitution proposed by Palsberg and Schwartzbach [21, 20] (henceforth denoted PS).

In the following section we examine the previous type systems and discuss their strengths, weaknesses, similarities, and differences. In section 3 we present the unified type system and its semantics, and indicate an efficient type-checking algorithm. Finally, in section 4 we show some example programs.

2 Previous 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 [3, 19], polymorphic types [18, 4], or F-bounded constraints [9, 8] 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 [5, 6].

In search for a better model, we examine some type systems designed for object-oriented languages with variables and assignments. The distinguishing features will be their notions of type, polymorphism, and genericity, see figure 1.

	SIMULA/BETA,C++	EIFFEL	TS	PS	Unified
Type	Set of all subclasses of a certain class		Finite set	Singleton	Apex set
Polymorphism	Inclusion			Equality	Inclusion
Genericity	Modifiable declarations	Parameterized classes		Type substitution	Class substitution

Figure 1: An overview of type systems.

2.1 Types

A *type* is an abstract description of a value. The values in object-oriented languages are either instances of classes or nil. Traditionally, for example in studies of functional languages, types are sets of values [4]. This view of types is not appropriate for object-oriented languages with variables and assignments because values are mutable. In the type systems SS, TS, and PS, the type of a value is a set of classes: if the value is an instance of a class, then that class must belong to the set; nil is a value of the empty set. The smaller a type is, the more precise is its description of a value. The type systems differ in which sets of classes are allowed. In SS, a type is the set of all subclasses of a certain class. In TS, a type is any finite set of classes, and in PS, a type is a singleton set, or—equivalently—simply a class.

Generalizing all three, the unified type system allows types to be so-called *apex* sets of classes. This essentially allows all "cross-combinations" of types from. the previous type systems; the technical definition is given in the following section. Note that the expressive power of SS is incomparable with that of TS; neither can emulate the other.

The SS and PS type systems enable separate compilation of classes; so does the unified type system. In TS, each complete program is analyzed separately, using abstract interpretation. This allows the generation of better code, but that code cannot be reused. Note that a TS *signature type* denotes a finite set of classes when analyzing a particular program. A signature in TS is essentially a class containing only method headings. A signature type is then the set of all subclasses of that class. Since programs are finite, the TS compiler can *expand* a signature type to a finite set of classes.

2.2 Polymorphism

Inclusion polymorphism allows objects to have more than one type [4]. When types are sets of classes, a natural notion of subtyping is *set inclusion.* In all the previous type systems, an object has both a declared type and its supertypes. For example, nil is a value of any type, since the empty set is included in any other. Notice also that in PS, subtyping reduces to equality of classes, since we only have singleton sets. In all the type systems, an assignment x:=e is legal if the object denoted by e is of the declared type of x. This is decidable on compile-time in PS, but not in the others where variables may hold instances of more than one class. Notice also that TS *axiomatizes* a relation that corresponds exactly to set inclusion; this is necessary because the type system involves type variables.

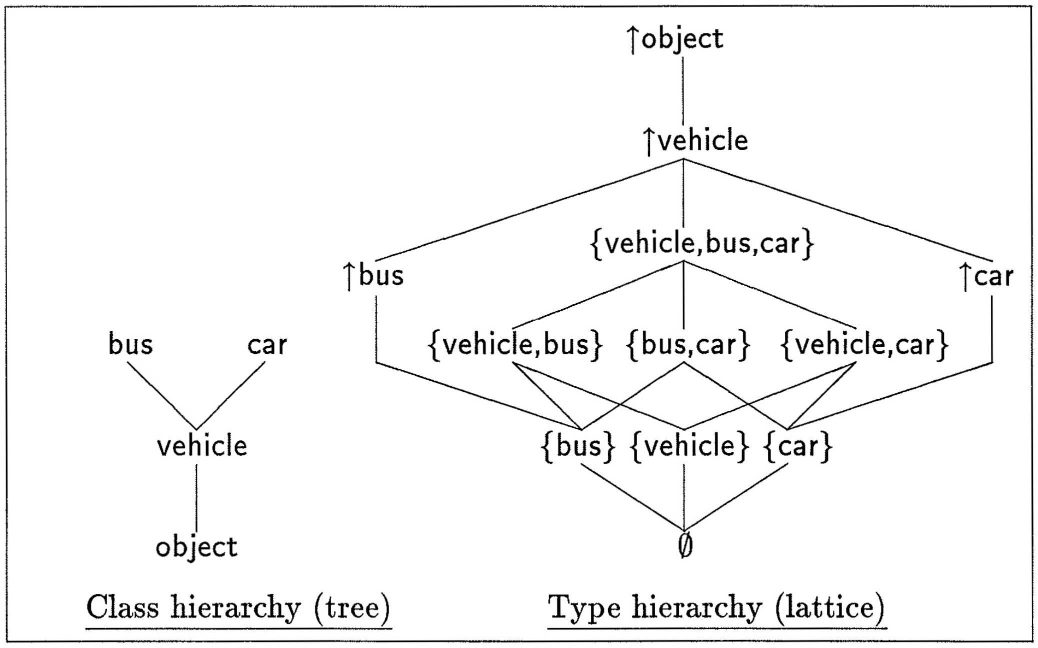

Figure 2: The class and type hierarchies (excerpts).

Following the previous type systems, the unified type system uses set inclusion as subtyping. For an illustration of the class and the type hierarchies, see figure 2. The class hierarchy is that discussed in one of the BETA group's papers [16]. Notice that we turn it "upside-down" in order to get the smallest class at the bottom. The figure uses the notation ↑C for the set of all subclasses of C.

Note that the class hierarchy is a *tree*, whereas the type hierarchy is a *lattice*. Note also that the BETA group interprets nil as an *instance* of an auxiliary class on top of the class hierarchy [16]. 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 of the empty set is more satisfactory: it reflects that nil is *not* an instance of any class, that it can *not* respond to any messages, and that it can be assigned to any variable.

When type checking an assignment x:=e in the unified type system, the compiler will find the declared type of x, $T(\mathsf{x})$, compute the static type of e, $T(\mathsf{e})$, and analyze whether the assignment is *safe*, *impossible*, or *possible*, see figure 3.

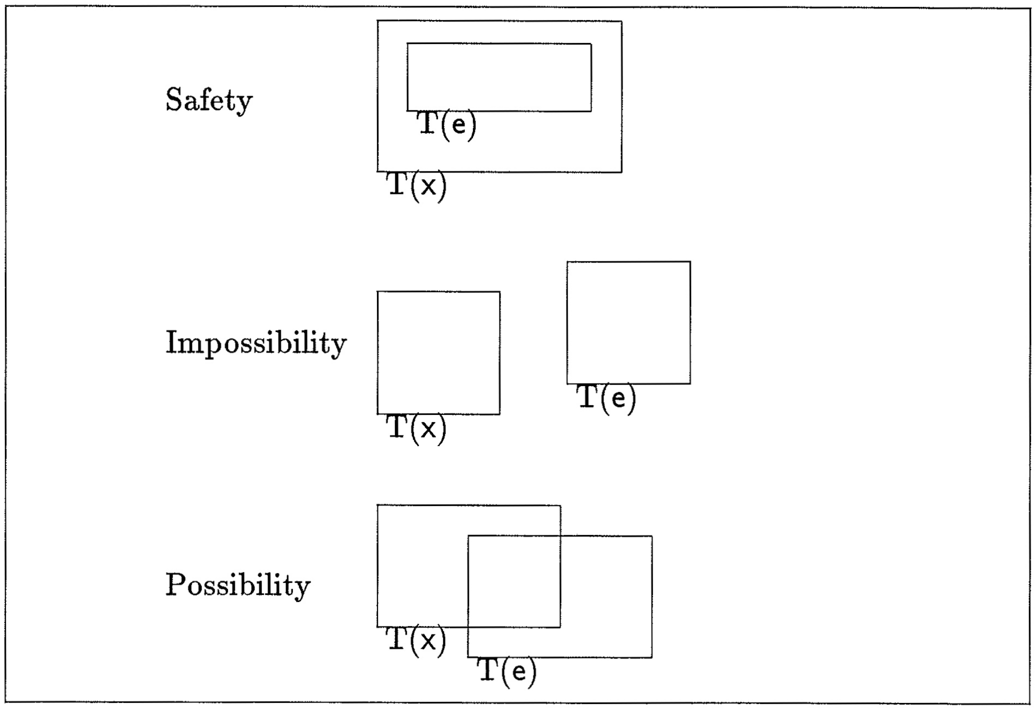

Figure 3: The three situations in type-checking.

The compiler must ensure that a variable only contains values of its type. This is always true after the execution of the assignment if $T(\mathsf{x}) \supseteq T(\mathsf{e})$;

we then call the assignment safe. Otherwise, there are two options for a compiler. It can either deem the assignment type-incorrect, as in TS, or analyze the situation a little further, as in SS and the unified type system. The further analysis will decide whether $T(\mathsf{x})$ and $T(\mathsf{e})$ are disjoint or not. If they are, then we have impossibility: the value will *never* have the type of the variable. If they are *not* disjoint, then we have possibility: the value may or may not have the type of the variable. The SS type systems all allow possible assignments but manage them differently. C++ simply ignores checking on run-time whether the assigned object has an appropriate type or not. EIFFEL requires the programmer to use a special syntax for possible assignments, and will then insert a run-time check in the code. SIMULA/BETA automatically discovers whether assignments are possible, and inserts run-time checks. Note that the TS compiler uses abstract interpretation to obtain as small a static type for e as possible. The effect of this is that some assignments that the SS type system view as possible can by the TS compiler be determined as either safe or impossible. In the unified type system, we will allow the automatic insertion of run-time checks in the code for possible assignments.

When type-checking a message send x.p(...), the compiler must ensure that each class in the static type of the object denoted by x implements a method p. This can be done by looking up the definition of a single class in SS and PS; in TS a case analysis is necessary.

2.3 Genericity

The above analysis must be strengthened in the presence of *genericity*, which allows the substitution of types in a class. One must ensure that type-correctness is preserved in all generically derived classes. This is guaranteed in the TS and PS systems, where types are finite, but not in the SS systems, where extra run-time checks may be needed.

Genericity can be obtained through parameterized classes as in TS or EIFFEL although they are inflexible since any class can be inherited but is not in itself parameterized. An alternative to parameterized classes is the use of *modifiable* (virtual) declarations [15], as in SIMULA/BETA, and C++. Note though, that individual conflicting modifications may yield type-incorrect subclasses of a type-correct class; this leads to a fair amount of run-time type-checking, which is superfluous if the resulting class is in fact type-correct. EIFFEL employs both parameterized classes and modifiable declarations, leading to the problems reported by Cook [7]. Palsberg and

Schwartzbach suggested the genericity mechanism *type substitution* which solves all these problems [21]. Type substitution is a subclassing concept that complement inheritance; any class is generic, can be "instantiated" gradually without planning, and has all of its generic instances as subclasses. Another pleasant fact is that type substitution and inheritance together form an orthogonal basis for a general subclass relation that captures type-safe code reuse [20]. Type substitution also avoids the type variables and second-order entities used in connection with parametrization; hence, it simplifies both the type system itself and the type-checking algorithm. The unified type system uses type substitution—now under the name *class substitution*, because types and classes are different. In figure 4 we summarize our conceptions of classes, values, and types.

Classes

- A class describes an implementation.
- Subclassing is reuse of class definitions.
- Genericity and inheritance are subclassing mechanisms.

Values

- Instances of classes are values.
- nil is also a value, but it is *not* an instance of any class.

- A type describes a behavior through a set of possible implementations.
- Subtyping is set inclusion.

Figure 4: Summary of classes versus types.

In the following section we formally present the unified type system. The combination of types as sets of classes, subtyping as set inclusion, and genericity as class substitution yields a general yet simple type system. For some example programs, see section 4.

3 A Unified Type System

The unified type system is *structural*: all classes and types are given *a priori*—independently of the programmer's definitions. Thus, the syntax for a class in a program will simply *denote* a class; it will not *create* a new class. This independency allows us to work in a precise mathematical framework.

3.1 Classes and Types

We shall give a simultaneous definition of two infinite sets of labeled, ordered, regular trees: $\mathcal{U}$ is the set of all classes, and $\mathcal{T}$ is the set of all types.

Definition 1 A *class* consists of a piece of untyped code annotated with *types* at various positions (declarations of variables and formal parameters). We shall denote such untyped code by variations of the symbol γ. A *type* is a set of classes. We shall only concern ourselves with *apex* sets, which can be denoted by pairs of lists of classes such as

$$[c_1, c_2, \ldots, c_k || d_1, d_2, \ldots, d_n]$$

where $k, n \geq 0$ and c_i, d_j are classes. The c_i's denote individual elements of the set, whereas the d_i's denote the class and all its subclasses. Viewing untyped code and the notation

$$[-, -, \ldots, - || -, -, \ldots, -]$$

as *labels*, we see that both types and classes are node-labeled, ordered trees. $\square$

Classes and types may occur in layers, as sketched in figure 5. The trees are not strictly bipartite, though, since the new expression involves a class and not a type.

We next present an independent definition of a binary subclass relation $\lhd$ on $\mathcal{U}$. It was presented in [21, 20] where it was shown that it generalizes the usual notion of inheritance and combines it with a general notion of genericity. These results, which we retain while introducing subtyping, are briefly reviewed in section 3.3.

Definition 2 If T is a labeled tree, then $\text{GEN}(T)$ is obtained from T by replacing all maximal, proper occurrences of T itself by the special label $\diamond$. This notion of a *generator* captures the recursive structure of a tree. $\square$

Definition 3 The ordering $\leq$ on labels is the usual prefix ordering. Note that this is discrete (trivial) on $[\ldots || \ldots]$ labels. $\square$

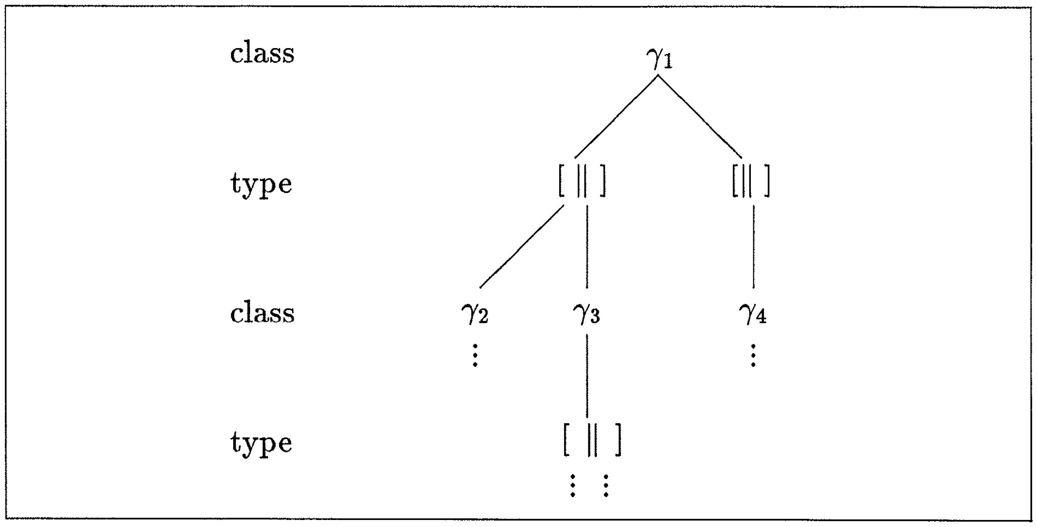

Figure 5: Classes and types.

Definition 4 Let T be a node-labeled ordered tree. 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. □

Definition 5 The relation $G_1 \lhd_G G_2$ on generators holds precisely when

- $\forall \alpha \in G_1 : G_1[\alpha] \leq G_2[\alpha]$
- $\forall \alpha, \beta \in G_1 : G_1 \downarrow \alpha = G_1 \downarrow \beta \Rightarrow G_2 \downarrow \alpha = G_2 \downarrow \beta$

The relation $T_1 \lhd T_2$ holds precisely when

- $\forall \alpha \in T_1 : \text{GEN}(T_1 \downarrow \alpha) \lhd_G \text{GEN}(T_2 \downarrow \alpha)$

We call T_1 the *superclass* and T_2 the *subclass.* □

Proposition 6 The relation $\lhd$ is a partial order on $\mathcal{U}$.
Proof: This follows since $\text{GEN}(T)$ unique given T, and $\leq, \Rightarrow$ themselves are partial orders. □

An example of the $\lhd$-order is illustrated in figure 6. The class with empty code, which we shall denote object, is the unique minimal class under $\lhd$.

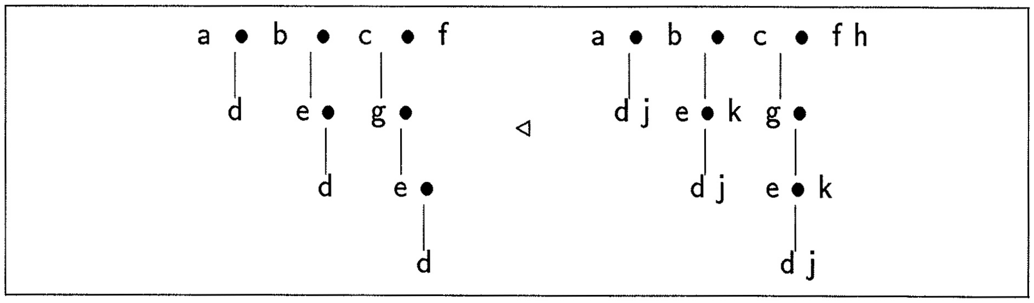

Figure 6: Two ◁-related trees.

Definition 7 Each $t \in \mathcal{T}$ corresponds to a set of classes as follows

$$[c_i||d_j] = \bigcup_i \{c_i\} \cup \bigcup_j \textsc{cone}(d_j)$$

where $\textsc{cone}(d) = \{d' \in \mathcal{U} \mid d \lhd d'\}$. Intuitively, a cone contains all the subclasses of its root class. Thus, we allow finite sets of classes combined with a finite union of ◁-cones rooted by classes. Such sets will be called *apex* sets. □

In comparison, the PS system allows only a singleton, the TS system allows only a finite set, and the SS systems allow only a single cone. For an illustration of an apex set, see figure 7.

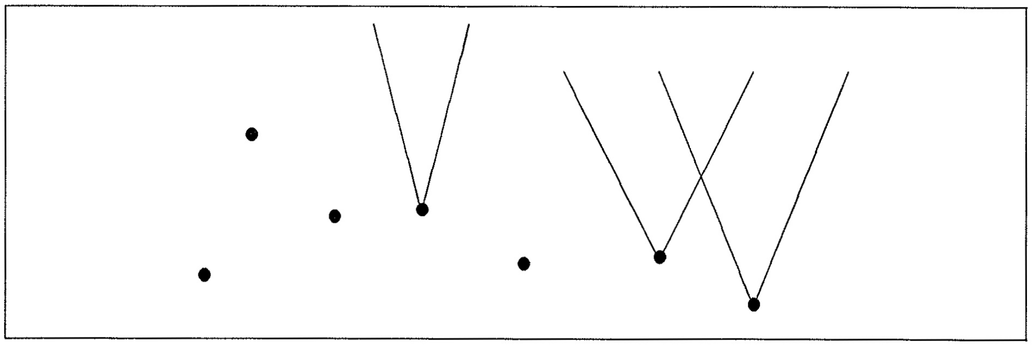

Figure 7: An apex set.

3.2 Type Expressions

For actual programming we offer a more convenient syntax for types.

Definition 8 The *type expressions* are generated by the grammar in figure 8 where $c \in \mathcal{U}$ is a constant. Each type expression τ denotes a unique set $[\![\tau]\!] \subseteq \mathcal{U}$, defined in the following manner

$$\boxed{\tau ::= c \mid \uparrow\tau \mid \tau_1 + \tau_2 \mid \mathsf{Anything} \mid \mathsf{Nothing}}$$

Figure 8: Syntax for type expressions.

- $[\![c]\!] = \{c\}$
- $[\![\uparrow\tau]\!] = \bigcup_{c \in [\![\tau]\!]} \text{CONE}(c)$
- $[\![\tau_(1) + \tau_2]\!] = [\![\tau_1]\!] \cup [\![\tau_2]\!]$
- $[\![\mathsf{Anything}]\!] = \mathcal{U}$
- $[\![\mathsf{Nothing}]\!] = \emptyset$

The syntax $\tau_1 + \tau_2$, Anything, and Nothing is inspired by TS, and $\uparrow \tau$ by BETA. □

Theorem 9 For any τ we have that $[\![\tau]\!]$ is an apex set.

Proof: We proceed by induction in the structure of τ.

- If $\tau = c$, then $[\tau] = [c||]$.
- If $\tau = \uparrow\tau'$ then we assume inductively that $[\![\tau']\!] = [c_i \,|\, d_j]$. It follows that
$$[\![\tau]\!] = \bigcup_{c \in [c_i||d_j]} \text{CONE } (c) = [||c_i, d_j]$$
- If $\tau = \tau_1 + \tau_2$ then we assume inductively that $[\![\tau_1]\!] = [c_i^1||d_j^1]$ and $[\![\tau_2]\!] = [c_i^2||d_j^2]$. Now, $[\![\tau]\!] = [c_i^1||d_j^1] \cup [c_i^2||d_j^2] = [c_i^1, c_i^2||d_j^1, d_j^2]$.
- If $\tau = \mathsf{Anything}$, then $[\![\tau]\!] = \mathcal{U} = [||\mathsf{object}]$.
- If $\tau = \mathsf{Nothing}$, then $[\![\tau]\!] = \emptyset = [|||]$.

Notice that this translation is computable. □

3.3 Genericity and Inheritance

This subsection reviews the results of [20, 21]. The subclass order a contains two suborders $\lhd_I$ and $\lhd_S$ that correspond respectively to ordinary *inheritance* and genericity in the form of *class substitution.*

Definition 10 The relations $\lhd_I$ and $\lhd_S$ as are defined as follows

- $C_1 \lhd_I C_2$ *iff* $C_1 \lhd C_2 \wedge \forall \alpha \in C_1 : C_1 \downarrow \alpha \neq C_1 \Rightarrow C_1[\alpha] = C_2[\alpha]$
- $C_1 \lhd_S C_2$ *iff* $C_1 \lhd C_2 \wedge C_1[\lambda] = C_2[\lambda]$

In inheritance only the code of the superclass itself (and its recursive occurrences) may be extended in the subclass; the type annotations must otherwise remain unchanged. In class substitution the untyped code of the superclass and the subclass must be the same; only the type annotations may change.

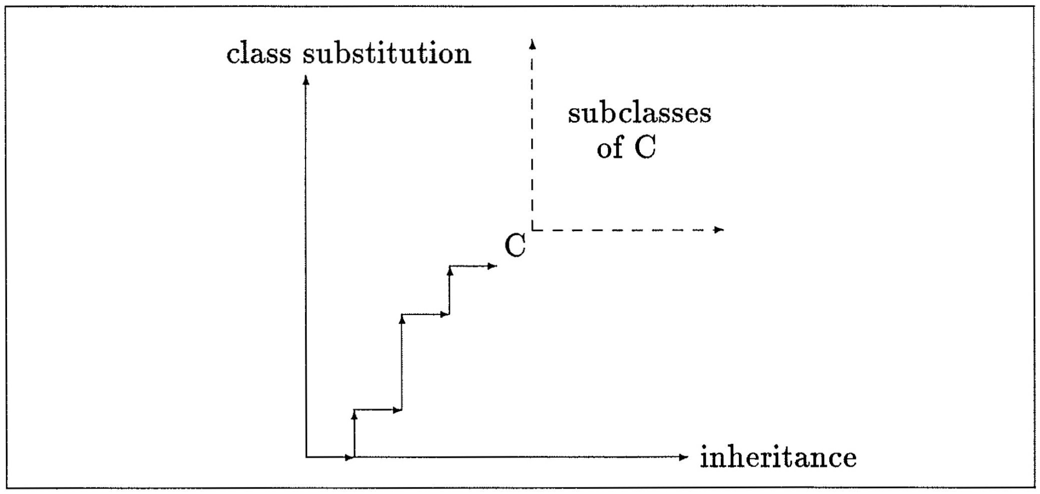

Figure 9: An orthogonal basis for subclassing.

We have the following fundamental result.

Theorem 11 $\lhd_I$ and $\lhd_S$ as form an order-theoretic orthogonal basis for $\lhd$.
Proof: See [20]. $\square$

The intuitive meaning of the above result is that any subclass can be obtained from the superclass in a finite number of inheritance and substitution steps. Furthermore, $\lhd_S$ as is the unique minima1 relation that together with $\lhd_I$ has this property, and vice versa. The situation is sketched in figure 9.

```
class C2 inherits C2
  γ
end C2
```

Figure 10: Syntax for inheritance.

Inheritance can be realized through the ordinary syntax. If C1 $\lhd_I$ C2 then C2 can be denoted as indicated in figure 10, where γ is the extra code in (the root label of) C2. We can develop a similar syntax for substitution.

Definition 12 We define $\leq$ on trees such that $T_1 \leq T_2$ *iff* $\forall \alpha \in T_1 : T_1[\alpha] \leq T_2[\alpha]$. Thus, it is a node-wise extension of the prefix ordering on labels. $\Box$

Definition 13 Let $\bar{A}$ and $\bar{B}$ be vectors of classes. We say that $\bar{A}$ is *consistent* with $\bar{B}$ when two trees with the same root labels and subtrees respectively $\bar{A}$ and $\bar{B}$ are $\lhd$-related. $\Box$

Theorem 14 For any consistent $\bar{A}, \bar{B}$ there is a unique $\leq$-minimal subclass of C in which all (visible) occurrences of A_i are substituted by B_i. This class is denoted by $C[\bar{A} \leftarrow \bar{B}]$.
Proof: See [20]. $\Box$

Note that $C[\bar{A} \leftarrow \bar{B}]$ is more sophisticated than the *textual* substitution of B_i for A_i; the latter need not produce a subclass.

Theorem 15 If C1 $\lhd_S$ C2 then C2 $=$ C1$[\bar{A} \leftarrow \bar{B}]$ for some consistent $\bar{A}, \bar{B}$.
Proof: See [20]. $\Box$

In [21] is argued for the pragmatic advantages of this mechanism.

3.4 Type-Correctness

A type annotated class may or may not be statically correct. The intention is that when a class is deemed statically correct, then the run-time error Message-not-understood will never occur [1]. Furthermore, this property must be preserved in all subclasses.

Based on the discussion in section 2, we propose the following typechecking rules.

Definition 16 A class C is *type-correct* when the following two kinds of checks are valid:

- Early checks: verify for all calls x.p(...) that a method p, with the proper number of parameters, is implemented by all the classes in the type of x in all subclasses of C.
- Inclusion checks: verify for all assignments x := e (and similarly for parameter passings) that the type of e is a subset of the type of x in all subclasses of C.

Type-correctness is a static property of program texts. $\square$

When we allow the introduction of run-time checks) then for every failed inclusion check we must further determine if the types of x and e are disjoint or not.

3.5 Type-C hecking Algorithm

Type-checking involves two kinds of checks. The early checks can be performed in the standard fashion, since they are valid for all classes in a cone *iff* they are valid for the root class; they will then also remain valid in all subclasses.

The inclusion checks, generalized to consider run-time checks) require a solution to the following problem.

Definition 17 The *Inclusion Check Problem* is when given two type expressions τ_1, τ_2 to decide

- SAFETY: $\forall$ consistent $\bar{A}, \bar{B} : [\![\tau_1]\!][\bar{A} \leftarrow \bar{B}] \subseteq [\![\tau_2]\!][\bar{A} \leftarrow \bar{B}]$

- IMPOSSIBILITY: $[\![\tau_1]\!] \cap [\![\tau_2]\!] = \emptyset$
- POSSIBILITY: $[\![\tau_1]\!] \cap [\![\tau_2]\!] \neq \emptyset$

These predicates are not disjoint. We want to find the first one that applies. Only in the third case are run-time checks required. Concerning safety, it is sufficient to consider $\lhd_S$-subclasses, as indicated. This true because inclusions obviously will be preserved in all $\lhd_I$-subclasses. $\square$

The following result shows that class substitution does not preserve non-trivial $\lhd$-relations.

Lemma 18 If $X \neq Y$ are classes, then there are classes $A \lhd B$ such that $X[A \leftarrow B] \not\lhd Y[A \leftarrow B]$.
Proof: If $X \not\lhd Y$ then we are done with any choice with $A = B$. If $X \lhd Y$ then we can find a minimal $\alpha \in X$ such that $X[\alpha] < Y[\alpha]$. Now, let $A = X \downarrow \alpha$ and choose B such that $Y \downarrow \alpha \lhd_I B$ and $Y[\alpha] < B[\lambda]$. Now $A \lhd B$ and $X[A \leftarrow B] \not\lhd [A \leftarrow B]$, since $X[A \leftarrow B][\alpha] \not\leq Y[A \leftarrow B][\alpha]$. $\square$

Hence, $\lhd$-relations can never be trusted to remain valid in subclasses.

From the proof of theorem 9 we see that the apex set $[\![\tau]\!]$ can be computed from τ. Thus, we need only provide a solution for apex sets directly. Furthermore, we need only decide the problem for singletons and single cones. This is because apex sets are finite unions of such, so the result follows from appropriate disjunctions and conjunctions. We now examine the four possible combinations.

- Singleton c_1, singleton c_2: In this case we can ignore substitutions, since they are all functional. Thus, safety holds when $c_1 = c_2$, and impossibility holds otherwise.
- Cone $\uparrow c_1$, singleton c_2: Since a cone is always infinite, we can never have safety. If $c_1 \not\lhd c_2$ then we have impossibility; otherwise, we have possibility.
- Singleton c_1, cone $\uparrow c_2$: If $c_1 = c_2$ then safety holds; otherwise, lemma 18 states that it does not. If $c_2 \not\lhd c_1$ then we have impossibility; otherwise, we have possibility.

- Cone $\uparrow c_1$, cone $\uparrow c_2$: If $c_1 = c_2$ then safety holds; otherwise, lemma 18 states that it does not. The intersection of $\uparrow c_1$ and $\uparrow c_2$ is exactly the set of $\lhd$-upper bounds of c_1 and c_2. If they have no upper bound then impossibility holds; otherwise, possibility holds.

The above analysis shows that we must decide three properties of classes: equality, the subclass relation $\lhd$, and the existence of $\lhd$-upper bounds. This is possible using finite state automata algorithmics; details are given in the appendix.

4 Examples

our example language is patterned after SMALLTALK-80 [11] and requires that variables and parameters are declared together with a *type*, given by the syntax presented in section 3.2. Inheritance and class substitution are the subclassing mechanisms; subtyping is set inclusion.

The example programs are reformulations of some taken from the HP ABEL group's paper on interfaces for strongly-typed object-oriented programming[2], the BETA group's paper on strong typing of object-oriented languages [16], Meyer's book on object-oriented software construction [17], and Cook's paper on problems in the EIFFEL type system [7]. These examples have been chosen to relate the unified type system to the critical issues discussed in the literature.

4.1 Point Classes

Consider the point classes in figure 11. A point object has an equal method for comparing itself with any object of a subclass of point. Class colorpoint inherits point; its objects can compare themselves with any object of a subclass of colorpoint. The code in these two classes is obviously typecorrect (no run-time checking is needed). In the example class we have declared four variables and performed four message sends. In a separate compilation, all four message sends require run-time checking. If only this particular program is considered and no other classes are assumed to be present, then the type $\uparrow$ point expands to point + colorpoint whereas $\uparrow$ colorpoint expands to colorpoint. Then only the first and the fourth message send require run-time checking; however, the compiled code will no longer be reusable.

```
class point
  var x,y: integer
  method equal(other: ↑ point) returns boolean
    return (x=other.x) and (y=other.y)
end point
class colorpoint inherits point
  var c: color
  method equal(other: colorpoint) returns boolean
    return (super.equal(other)) and (c=other.c)
end end colorpoint
class example
  var p, p': ↑ point
  var c, c': ↑ colorpoint
  ... p.equal(p')
  ... c.equal(c')
  ... p.equal(c')
  ... c.equal(p')
end example
```

Figure 11: Point classes.

The HP Abel group argues that a separate hierarchy of interfaces is necessary for the correct type-checking of these classes [2]. The example shows, however, that the notion of types as sets of classes is appropriate: it is simple, general, and leads to an efficient type-checking algorithm.

As noted by the Beta group [16], if p and p' instead are declared as of type point (a singleton type), and q and q' as of type colorpoint then no run-time checking is needed. In this case, the first three message sends are type-correct whereas the forth is type-incorrect.

4.2 List Classes

Consider next the heterogeneous list classes and the (insertion) sort procedure in figure 12. Any object of a subclass of recordlist can be sorted; the resulting object is of the same class. All the code in these three classes is type-correct (no run-time checking is needed). To see this, consider for instance the sort procedure. The expression empty evaluates to a boolean, by declaration. The expression self evaluates to the receiver object which is of the declared result class, because of recursion. The expression tail.sort evaluates to an object of the same class as the receiver. In this class, head is an object of a subclass of record, thus the message send (tail.sort).insert(head) is type-correct and evaluates to an object of the same class as the receiver, which is the declared result class.

Notice the application of class substitution in the definition of recordlist. In fact, list acts like a parameterized class but is just a class, not a second-order entity. If the program should have been written using parameterized classes, then the creation of recordlist must be planned ahead by writing a parameterized list class, and thereby also introducing the problems reported by Cook [7].

The program could also have been written using the modifiable (virtual) declarations of SIMULA/BETA. sing this approach, the declarations of head and x would both be modified in class recordlist to have type $\uparrow$record. If the declarations of head and x can be modified separately, then run-time checking is required in all six places where either head or x (or both) are used. This clearly yields poorer run-time performance than with our approach, which does not introduce run-time checks into this program.

4.3 Cook's Example

Let us finally reexamine (a reformulation of) one of the EIFFEL programs that Cook provided in his paper on problems in the EIFFEL type system [7], see figure 13. In the example class we have declared two variables, and performed an assignment and a message send. This leads to a runtime error, as noted by Cook, because get in the son object will try to access the extra procedures of its argument which does not exist. In a separate compilation, the assignment p:=s needs a run-time check because parent may in a subclass be substituted by something greater than or incomparable with son. Also the message send p.get(**new** parent) needs a run-time check (which will fail).

In a traditional compilation, assuming absence of other classes, the type $\uparrow$parent expands to parent+son whereas $\uparrow$son expands to son. Then the assignment is statically type-correct (but the message send still requires run-time checking). An abstract interpretation of the two expressions, in the style of Graver [13], would reveal that p.get(**new** parent) has no chance of being type-correct, and could thus deem the program type-incorrect .

```
class record
  var key: integer
end record
class list
  var empty: boolean
  var head: ↑ integer
  var tail: list
  method cons(x: ↑ object) returns list
    var result: list
    result:=new list
    result.empty:=false
    result.head:=x
    result.tail:=self
    return result
end list
class resordlist inherits list[object ← record]
  method insert(x: ↑ record) returns recordlist
    if empty or else x.key < head.key
    then return self.cons(x)
    else return (tail.insert(x)).cons(head)
  method sort returns recordlist
    if empty
    then return self
    else return (tail.sort).insert(head)
end recordlist
```

Figure 12: List classes and a sort procedure.

5 Conclusion

The unified type system is more expressive than previous type systems for ob ject-oriented languages with assignments. Yet, its type-checking algorithm is conceptually simple, due to the chosen representations of classes and types.

As a continuation of this work, we want to investigate techniques for type inference that would construct the type annotations for completely untyped programs. Subsequently, such typed programs can be subjected to abstract interpretation for the purpose of removing superfluous runtime checks. Since

```
class parent
  method get(arg: parent) returns integer
    return 0
end parent
class son inherits parent
  method extra returns integer
    return 0
  method get(arg: son) returns integer
    return arg.extra
end son
class example
  var p: ↑ parent
  var s: ↑ son
  . . .
  p:=s
  p.get(new parent)          (* run-time error *)
end example
```

Figure 13: Cook's example.

our type system caters for run-time checks, it provides a clean division between the two tasks.

A Algorithms on Classes

To be able to decide equality, $\lhd$, and the existence of $\lhd$-upper bounds, we provide a finite representation of classes and types.

Proposition 19 Every regular, node-labeled 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. $\Box$

Of course, these automata can be constructed directly from the program text.

Lemma 20 Equality of regular, node-labeled trees is decidable.
Proof sketch: A variation of the standard algorithm for language equality of automata solves this problem in pseudo-linear time. □

Lemma 21 The relation $\lhd$ on regular, node-labeled trees is decidable.
Proof sketch: If the two automata are initially minimized, then the relation $\lhd_G$ can be decided in linear time during a graph traversal. An automaton of size n can be minimized in time $O(n \log n)$. Since we have at most n distinct generators, decidability follows. □

To decide the existence of an upper bound, it would be natural to compute the least upper bound. Unfortunately, this may not exist.

Lemma 22 There exist classes with two distinct minimal $\lhd$-upper bounds.

Proof: Look at the classes in figure 14. The classes $\mathsf{U1}$, $\mathsf{U2}$, and $\mathsf{U3}$ are all minimal upper bounds of the classes $\mathsf{C1}$ and $\mathsf{C2}$. In particular, note that $\mathsf{U3}$ is not $\lhd$-less than $\mathsf{U1}$ or $\mathsf{U2}$. □

However, there is an upper bound that is least in another ordering.

Lemma 23 If two classes have a $\lhd$-upper bound, then they have a unique $\leq$- least one.
Proof: We must simply show that $\lhd$-upper bounds are closed under $\leq$-greatest lower bounds, $\sqcap_{\leq}$. For any two classes U_1 and U_2 we see that $U_1 \sqcap_{\leq} U_2$ exists and is obtained by node-wise largest common prefix of labels. Assume that U_1 and U_2 are both $\lhd$-upper bounds of C_1 and C_2. We must show that so is $U = U_1 \sqcap_{\leq} U_2$. Clearly, we have that $C_i \leq U$. Assume now that $C_i \downarrow \alpha = C_i \downarrow \beta$. Since C_i is $\lhd$-related to both U_1 and U_2, we have that

$$U \downarrow \alpha = U_1 \downarrow \alpha \sqcap_{\leq} U_2 \downarrow \alpha = U_1 \downarrow \beta \sqcap_{\leq} U_2 \downarrow \beta = U \downarrow \beta$$

It follows that $C_i \lhd U$, as required. □

In figure 14, the $\leq$-least $\lhd$-upper bound of $\mathsf{C1}$ and $\mathsf{C2}$ is $\mathsf{U3}$.

Lemma 24 There is an algorithm to compute the $\leq$-least $\lhd$-upper bound of two classes, or decide that none exists.
Proof sketch: Starting again with minimized automata, this can be done

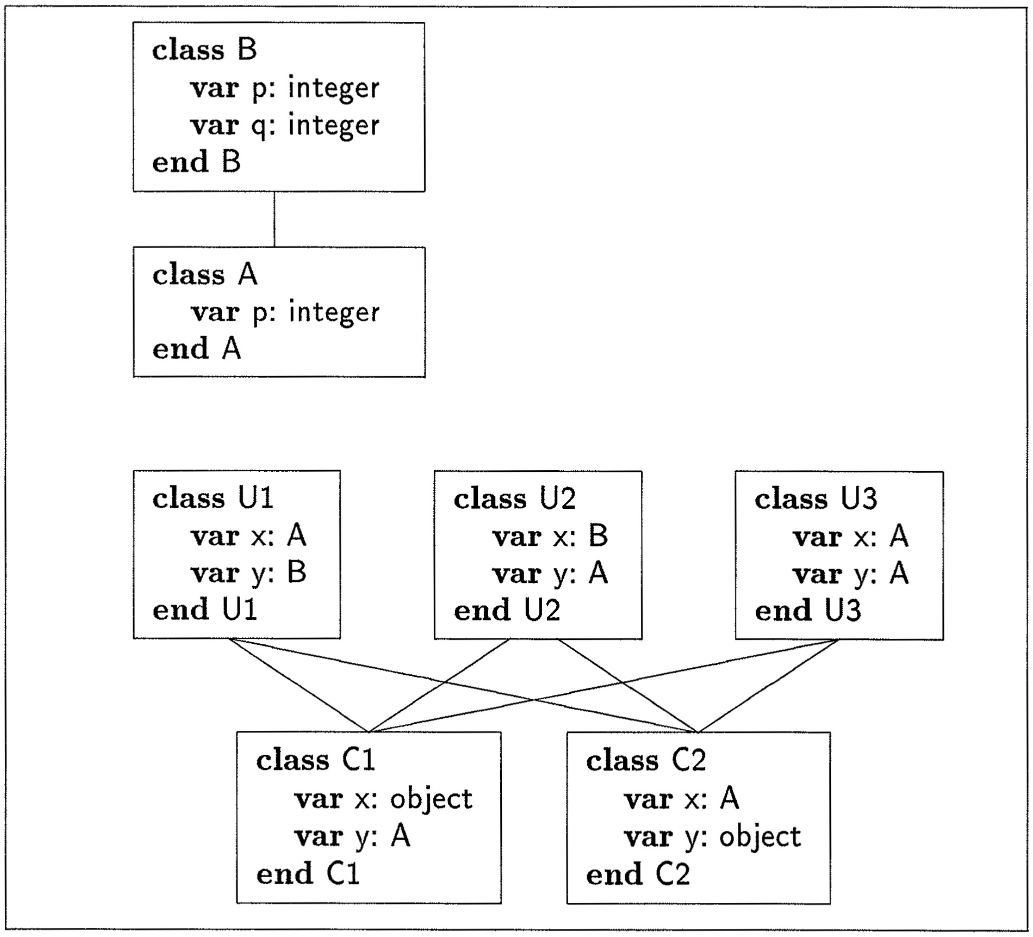

Figure 14: Distinct minimal upper bounds.

essentially by a straight-forward recursive algorithm using dynamic programming. The time complexity is bounded by the required size of the dynamic table, which may become exponential in extreme cases. □

References

[1] 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. ACM Press, January 1982.

[2] 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.* ACM, 1989.

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

[4] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4), December 1985.

[5] Luca Cardelli. Typeful programming. Technical report, Digital Equipment Corporation, 1989.

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

[7] William Cook. A proposal for making Eiffel type-safe. In *Proc. ECOOP'89, European Conference on Object-Oriented Programming*, 1989.

[8] William Cook, Walter Hill, and Peter Canning. Inheritance is not subtyping. In *Seventeenth Symposium on Principles of Programming Languages.* ACM Press, January 1990.

[9] William R. Cook, Walter L. Hill, and Peter S. Canning. F-bounded polymorphism for object-oriented programming. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, 1989.

[10] O. J. Dahl, B. Myhrhaug, and K. Nygaard. Simula 67 common base language. Technical report, Norwegian Computing Center, Oslo, Norway, 1968.

[11] A. Goldberg and D. Robson. *Smalltalk-80—The Language and its Implementation.* Addison-Wesley, 1983.

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

[13] 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.

[14] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. The Beta programming language. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, 1987.

[15] Ole L. 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.* ACM, 1989.

[16] 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*, 1990.

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

[18] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 1978.

[19] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Seventeenth Symposium on Principles of Programming Languages.* ACM Press, January 1990.

[20] Jens Palsberg and Michael I. Schwartzbach. Genericity And Inheritance. Computer Science Department, Aarhus University. PB-318, 1990.

[21] 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*, 1990.

[22] B. Stroustrup. *The* C++ *Programming Language.* Addison-Wesley, 1986.