ISSN 0105-8517

Genericity and Inheritance

Jens Palsberg
Michael I. Schwartzbach

DAIMI PB – 318
July 1990

ISSN 0105-8517

PB – 318 Palsberg & Schwartzbach: Genericity and Inheritance

Genericity And Inheritance

Jens Palsberg[1] **Michael I. Schwartzbach**[2]

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

Abstract

We present *type substitution* as a new genericity mechanism for object-oriented languages. It is a subclassing concept on the same footing as inheritance, and is more flexible than parameterized classes. We prove that type substitution and inheritance together form an orthogonal basis for a general subclass relation that captures type-safe code reuse. Thus, genericity and inheritance are independent, complementary components of a unified concept. Our result is obtained in a novel model of classes which encompasses classes as types and assignments.

1 Introduction

Genericity and inheritance are different mechanisms for type-safe code reuse in object-oriented languages [21]. *Genericity* allows the substitution of types in a class, whereas *inheritance* allows the construction of subclasses by adding variables and procedures, and by replacing procedure bodies [14, 32]. The general syntax for inheritance is given in figure 1. Usually, *parameterized* classes are offered as the genericity mechanism, exemplified by Eiffel [22], Trellis/Owl [27], and Demeter [17], by the Ada construct of generic packages [13], and by parameterized CLU clusters [18].

A parameterized class is a second-order entity which is instantiated to specific classes when actual type parameters are supplied. Such instantiation is less flexible than inheritance, since any class can be inherited

[1]E-mail address: palsberg@daimi.dk
[2]E-mail address: mis@daimi.dk

```
class C inherits P
   ... more code ...
end
```

Figure 1: Inheritance.

$$\mathsf{P}[V_1, \ldots, V_n \leftarrow W_1, \ldots, W_n]$$

Figure 2: Type substitution.

but is not in itself parameterized. In other words, code reuse with parameterized classes requires planning; code reuse with inheritance does not.

This paper introduces *type substitution* as a new genericity mechanism. The general syntax for type substitution is given in figure 2. It is a *specification* of a type-correct subclass of P in which occurrences of the classes V_i have been substituted by subclasses W_i. As opposed to parameterized classes, type substitution *cannot* be explained as textual substitution because the result would not necessarily be type-correct. Instead, we prove the existence of a unique most general such subclass, which is taken to be the meaning of the specification. We also indicate efficient algorithms for an implementation.

Type substitution is a subclassing mechanism on the same footing as inheritance, and it involves no second-order entities or type variables. This gives many pragmatic advantages: any class is generic, can be "instantiated" gradually without planning, and has all of its generic instances as subclasses.

We also define a new model of classes which, unlike more traditional models, encompasses *classes as types* and *assignments*. A class is modeled as a possibly infinite, regular, node-labeled, ordered tree. The labels correspond to the untyped code of classes.

In this model, we prove that type substitution and inheritance together form an orthogonal basis for a general subclass relation that captures type-safe code reuse. We also prove that any subclass in a unique way can be obtained by one type substitution followed by an application of inheritance. Thus, genericity and inheritance are independent, complementary components of a unified concept. This reconciles genericity and inheritance [21], and shows that they are two dimensions [31] of subclassing, see figure 3.

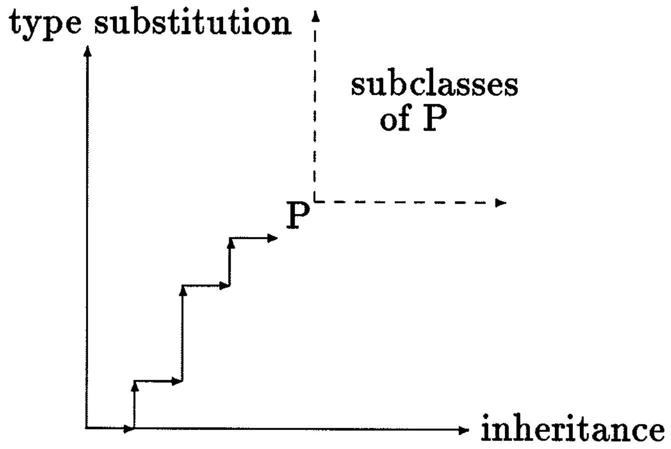


Figure 3: Two dimensions of subclassing.

In the following section we informally discuss the details and use of type substitution. In section 3 we formally define a class model, a general subclass relation, type substitution, and inheritance. Finally, in section 4 we prove our results.

2 Type Substitution

As an example of the use of type substitution, consider the monoid classes in figure 4.

In the class monoid[object $\leftarrow$ boolean] the type of value (declared in monoid) is substituted by boolean. Class booleanmonoid then implements the inherited procedures appropriately. Notice that monoid is a superclass of monoid[object $\leftarrow$ boolean] which again is a superclass of booleanmonoid. In other words, because monoid is in itself generic, we can "instantiate" it to monoid[object $\leftarrow$ boolean] which then can be inherited. Obviously, booleanmonoid cannot be obtained from monoid using only either type substitution or inheritance.

Using parameterized classes the instantiation of monoid must be planned

```
class monoid
  var value: object
  proc plus(other: monoid)
  proc zero
end
class booleanmonoid inherits monoid[object ← boolean]
  proc plus
    begin value:=(value or other.value) end
  proc zero
    begin value:=false end
end
```

Figure 4: Monoid classes.

```
class C1
  var x: object
end
let C2 = C1[object ← integer]
class D1
  var c: C1
  proc p(arg: object)
    begin c.x:=arg end
end
let D2 = D1[C1 ← C2]
```

Figure 5: Type substitution is *not* textual substitution.

ahead. An alternative to parameterized classes is the use of *modifiable* declarations, exemplified by the Beta notion of *virtual* class attributes [16, 19]. In the example, the declaration of object in monoid would be made modifiable; consequently, in booleanmonoid it could be modified to boolean. This technique also allows generic instances to be subclasses, but it still requires planning. Furthermore, individual conflicting modifications may yield type-incorrect subclasses; this leads to a fair amount of run-time type-checking [20], which is superfluous if the resulting class is in fact type-correct.

As an example of why type substitution cannot, in general, be explained as textual substitution, consider figure 5. If we try to obtain D2 as D1 with C1 textually substituted by C2, then the assignment in procedure p

```
proc swap(inout x,y: object)
   var t: object
   begin t:=x; x:=y; y:=t end
```

Figure 6: A polymorphic procedure.

involves different types: an integer and an object. Using type substitution, D2 is the most general type-correct subclass of D1 with C1 substituted by C2. This means that also object is substituted by integer. In fact, an equivalent specification of D2 is D1[object ← integer]. Of course, we must avoid inconsistent specifications, which cannot be realized. For example, one cannot substitute object by both boolean and integer. A formal definition of consistency is given in a later section.

Type substitution can also be the basis of another construct: polymorphic procedures declared outside classes, as illustrated in figure 6. Such procedures can be called independently of objects, allowing a symmetric programming style. When such a procedure is called, the compiler will (from the actual and formal parameter types) infer the required type substitution, verify that it is consistent, and perform it on a notional class that contains only the procedure. In the example, swap can be called with any two objects of the same type.

For further examples of how to program with type substitution, see [25].

3 The Class Model

To obtain a framework in which to prove our results, we provide a model of classes.

3.1 Motivation

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 for example subtypes [3, 24], polymorphic types [23, 4], or F-bounded constraints [9, 7] in the description of inheritance.

These models, however, do not support two common features of object-oriented languages, e.g. Simula [12], C++ [30], and Eiffel [22].

- The use of *classes* as types. In the previous models, types do not represent classes, but interfaces [2].
- *Variables* and *assignments*. In the previous models, variable types have no non-trivial subtypes. More specifically, as noted by Cardelli [5], a variable can be understood as a fetch-store pair for a hidden location, i.e. **Var**(T) abbreviates

  Tuple fetch():T, store(:T):Ok **end**

  Since T appears in both positive and negative positions, the type **Var**(T) is only related to itself.

We now define a novel model that supports both features. Since inheritance is about extending *code*, and type substitution is about changing *type annotations*, the model explicitly involves these concepts.

3.2 Class Denotations

Source code for classes will be denoted by variations of the symbol γ. They may contain variable and procedure declarations, procedure calls, and assignments. Code segments are sequences of declarations, which are prefix ordered such that $\gamma \leq \gamma\gamma'$.

Source codes are *untyped*, but they have a number of *positions* in which type annotations may be inserted. If $\gamma \leq \gamma'$ then the positions in γ is a subset of those in γ'. For example, the untyped code for the class monoid in figure 4 is

var value: • **proc** plus(other: •) **proc** zero

where • indicates a type position. The code

var value: •

is an example of a prefix of the above. The rules to check whether source code is *type-correct* are

- Early checks: verify for all calls x.p(···) that a procedure p is implemented by the declared type of x.

- Equality checks: verify for all assignments and parameter passings that the two declared types are equal.

Correctness guarantees that the run-time error `Message-not-understood` will never appear; this is the traditional statement of correctness [1]. We do not deal with *heterogeneous* variables which may hold objects of not only their declared class but also its subclasses; this concept invariably requires run-time type-checking [25, 20].

Since types are classes, we can provide a unique *denotation* for a typed class; it is a node-labeled, ordered tree. The root is labeled by the code of the class, and for each type position in the code there is a subtree which is the denotation of the corresponding class. Because of recursion these trees may be infinite. However, since a program can only employ finitely many classes, we can assume that all denotations are *regular*, i.e. they only contain finitely many *different* subtrees [11, 28]. We denote by $\mathcal{U}$ the collection of denotations.

Named classes such as integer and boolean merely abbreviate denotations, and object is simply the singleton tree whose label is the empty code sequence.

Note that the standard dynamic semantics of method lookup [8, 26, 6] can be based exclusively on these denotations. Thus we need only explain inheritance and type substitution by their effect on denotations; their dynamic semantics can then be inferred.

Definition 3.1: 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. The i'th immediate subtree of T is denoted by $T(i)$.

Proposition 3.2: Every regular, 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. □

These automata provide finite representations of denotations. All later algorithms on trees will in reality work on such automata.

Definition 3.3: A tree is *extended* when its leaves may contain the special label $\Diamond$. There is a partial order $\sqsubseteq$ on extended trees such that $G_1 \sqsubseteq G_2$ *iff* G_1 can be obtained from G_2 by replacing some subtrees by $\Diamond$. The unique smallest extended tree is $\Diamond$ itself. If G is an extended tree, then $G(T)$ is obtained from G by replacing all occurrences of $\Diamond$ with T.

Proposition 3.4: For every tree T there is a unique $\sqsubseteq$-smallest extended tree $\text{GEN}(T)$ such that T is the $\sqsubseteq$-smallest fixed point of $\text{GEN}(T)$. We call this the *generator* of T.
Proof: $\text{GEN}(T)$ is obtained from T by replacing all maximal proper occurrences of T with $\Diamond$. For details on such fixed points, see [10]. □

Note that there is a 1-1 correspondence between trees and generators. We call T *recursive* when at least one $\Diamond$-label occurs in $\text{GEN}(T)$, i.e. when $T \neq \text{GEN}(T)$. The generator may be an infinite tree when T contains a recursive class other than itself.

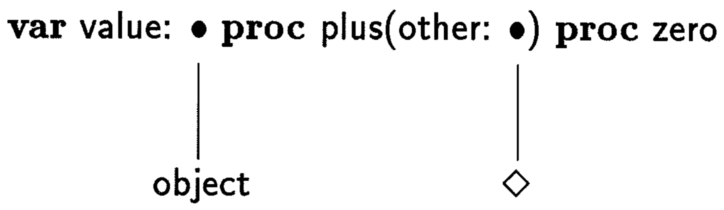


Figure 7: An example generator.

Figure 7 shows the generator of the recursive monoid class from figure 4. Its denotation is an infinite, linear tree.

3.3 Subclassing

We can now define a general subclass relation $T_1 \lhd T_2$ on $\mathcal{U}$ that captures type-safe code reuse; here, T_1 is the superclass and T_2 is the subclass. By *code reuse* we mean that the code of the superclass is a prefix of the code of the subclass. By *type-safe* we mean that type-correctness is preserved, i.e. we must guarantee that all early and equality checks will still hold in the subclass.

Definition 3.5: The relation $G_1 \lhd_G G_2$ on generators holds precisely when

- $\forall \alpha \in G_1 : \; G_1[\alpha] \leq G_2[\alpha]$ (*monotonicity*)
- $\forall \alpha, \beta \in G_1 : \; G_1 \downarrow \alpha = G_1 \downarrow \beta \;\Rightarrow\; G_2 \downarrow \alpha = G_2 \downarrow \beta$ (*stability*)

The relation $T_1 \lhd T_2$ holds when $\text{GEN}(T_1) \lhd_G \text{GEN}(T_2)$.

Monotonicity means that code can only be extended; stability means that equal types must remain equal.

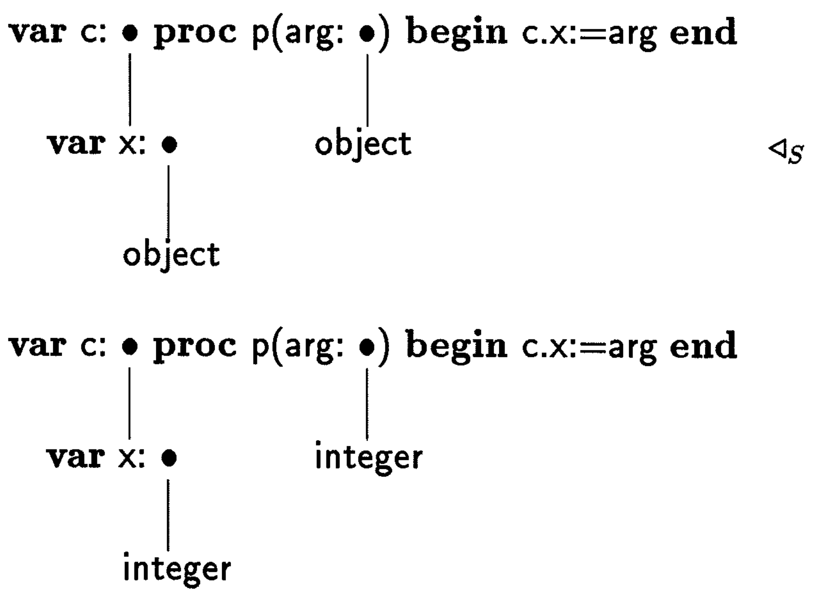
Notice that the relation is defined in terms of the generators of class denotations. If a similar concept of generators is introduced in the type system underlying F-bounded polymorphism [9, 7], then an F-bound is the ordinary subtype ordering of generators.

Finally, we define two relations $\lhd_I, \lhd_S \subseteq \lhd$ which capture inheritance and type substitution.

Definition 3.6: The relations $\lhd_I$ and $\lhd_S$ are defined as follows

- $T_1 \lhd_I T_2$ *iff* $T_1 \lhd T_2 \;\wedge\; \forall \alpha \in T_1 : \; T_1 \downarrow \alpha \neq T_1 \Rightarrow T_1[\alpha] = T_2[\alpha]$
- $T_1 \lhd_S T_2$ *iff* $T_1 \lhd T_2 \;\wedge\; T_1[\lambda] = T_2[\lambda]$

Informally, if $T_1 \lhd_I T_2$ then T_2 contains more code, and if $T_1 \lhd_S T_2$ then T_2 contains larger types. Figure 8 illustrates the $\lhd_S$-relationship between the denotations of the classes D1 and D2 from figure 5.



var c: • proc p(arg: •) begin c.x:=arg end
var x: •
object
object
◁S
var c: • proc p(arg: •) begin c.x:=arg end
var x: •
integer
integer

Figure 8: An example of $\triangleleft_S$.

4 Results

The subclass relation $\triangleleft$ is a partial order, and any class has only finitely many superclasses.

Proposition 4.1: The relation $\triangleleft$ is a decidable, partial order. Furthermore, for any T the set $\{T' \mid T' \triangleleft T\}$ is finite.
Proof: That $\triangleleft_G$ is a partial order follows immediately from the fact that $\leq$ and $\Rightarrow$ are partial orders. An algorithm in [29] decides $\triangleleft_G$ in time $O(n \log n)$, where n is the size of the corresponding automata. Since generators can easily be computed, $\triangleleft$ is decidable. For the finiteness of $\{T' \mid T' \triangleleft T\}$, let A be an automaton for T. If T' is strictly $\triangleleft$-smaller than T then it has an automaton which is obtained from A by deleting states and transitions, and reducing labels to proper prefixes. There can only be finitely many such automata.$\square$

A partial order may have an orthogonal basis, in the following sense.

Definition 4.2: Let P be a partial order. We write $Q \perp_P R$, if Q and R are partial orders such that $Q \cap R = id_P$ and $(Q \cup R)^* = P$. When

1) $Q \perp_P R$
2) $Q' \perp_P R \Rightarrow Q \subseteq Q'$
3) $Q \perp_P R' \Rightarrow R \subseteq R'$

we call Q, R an *orthogonal basis* for P. This generalizes the notion of *basis* in [15].

For example, if P_1 and P_2 are partial orders then $P_1 \times id_{P_2}$ and $id_{P_1} \times P_2$ form an orthogonal basis for $P_1 \times P_2$. We can now state our main theorem.

Theorem 4.3: $\lhd_I, \lhd_S$ is an orthogonal basis for $\lhd$, and $\forall T_1 \lhd T_2\ \exists! T' : T_1 \lhd_S T' \lhd_I T_2$.
Proof: See appendix A. □

Notice that in general there does *not* exist T'' so that $T_1 \lhd_I T'' \lhd_S T_2$. This is because the extra code in T_2 may exploit the larger types.

The general syntax for type substitution requires that the argument vectors are consistent, in the following sense.

Definition 4.4: Let $\vec{V}, \vec{W} \in \mathcal{U}^n$ be vectors of class denotations. We call $\vec{V}$ and $\vec{W}$ *consistent* when

$$V_i \lhd W_i \ \wedge\ \forall \alpha, \beta :\ V_i{\downarrow}\alpha = V_j{\downarrow}\beta \ \Rightarrow\ W_i{\downarrow}\alpha = W_j{\downarrow}\beta$$

Informally, $\vec{W}$ must be pointwise $\lhd$-larger than $\vec{V}$, and leave equal classes equal.

The set of all possible realizations of $\vec{V} \leftarrow \vec{W}$ can then be defined as follows.

Definition 4.5: Let $T \in \mathcal{U}$ be a class and $\vec{V}, \vec{W} \in \mathcal{U}^n$ consistent vectors of classes. We then define

$$\textsc{sub}(T,\vec{V},\vec{W}) = \{t \mid T \triangleleft_S t \ \wedge\ \forall i,j:\ \textsc{gen}(T)(i)=V_j \ \Rightarrow\ \textsc{gen}(t)(i)=W_j\}$$

Informally, these are the subclasses of T in which occurrences of V_j have been substituted by W_j.

Note that only immediate subtrees of the root of T are potential occurrences; this captures the idea that occurrences must be *visible.*

This set has an element with smaller types than any other.

Theorem 4.6: The set $\textsc{sub}(T,\vec{V},\vec{W})$ contains a unique nodewise $\leq$-smallest element which we denote $T(\vec{V} \leftarrow \vec{W})$.
Proof: See appendix B. □

This element is taken to be the meaning of the specification $T[\vec{V} \leftarrow \vec{W}]$.

Theorem 4.7: If $T \triangleleft_S t$ then $t = T(\vec{V} \leftarrow \vec{W})$ for some $\vec{V}$, $\vec{W}$.
Proof: Assume that T has n immediate subtrees; then so does t, since they have the same root label. We now have

$$t = T(T(1), T(2), \ldots, T(n) \leftarrow t(1), t(2), \ldots, t(n))$$

Note that monotonicity and stability between T and t ensure consistency of the argument vectors. □

Hence, all $\triangleleft_S$-related subclasses can be expressed in this manner.

Theorem 4.8: Consistency is decidable, and the automaton for the class $T(\vec{V} \leftarrow \vec{W})$ can be constructed from the automata for T, V_j, and W_j.
Proof: Consistency reduces to $\triangleleft_G$ on two trees with equal roots and subtrees respectively $\{V_j\}$ and $\{W_j\}$. Hence, decidability follows from lemma 4.1. The construction of the automata is easily seen from the proof of theorem 4.7. □

This shows that all relevant properties of $T(\vec{V} \leftarrow \vec{W})$ can be decided from the specification itself.

5 Conclusion

A new genericity mechanism for typed object-oriented languages was defined; it was proved to be the natural, orthogonal complement of inheritance in a class model supporting classes as types and assignments.

Acknowledgements: The authors thank Peter Mosses, Flemming Nielson, and Ole Lehrmann Madsen for helpful comments on a draft of the paper.

Appendix A: Proof of Theorem 4.3

Lemma A.1: $\lhd_I, \lhd_S$ are both partial orders, and $\lhd_I \cap \lhd_S = id$.
Proof: Clearly, $\lhd_S$ is a partial order. The extra condition on $T_1 \lhd_I T_2$ simply means that for every $\alpha \in \text{GEN}(T_1)$ we have $\text{GEN}(T_1)[\alpha] = \text{GEN}(T_2)[\alpha]$, 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 A.2: $\forall T_1 \lhd T_2 \; \exists! T' : T_1 \lhd_S T' \lhd_I T_2$.
Proof: Suppose $T_1 \lhd T_2$. Then $\text{GEN}(T_1) \lhd_G \text{GEN}(T_2)$ by definition. Let γ be the root label of $\text{GEN}(T_1)$. Then the root label of $\text{GEN}(T_2)$ must look like $\gamma\gamma'$. Let T' be obtained from $\text{GEN}(T_2)$ by removing the γ'-part of the root and the subtrees that occupy its positions. Since subtrees with the same address in $\lhd_G$-related trees also will be $\lhd_G$-related, it follows that $\text{GEN}(T_1) \lhd_G T'$. But since they both have root label γ, we also have $T_1 \lhd_S T'$. It is trivially the case that $T' \lhd_I T_2$, so we have shown that $T_1 \lhd_S T' \lhd_I T_2$.

For the uniqueness of T', suppose we also have $T_1 \lhd_S T'' \lhd_I T_2$. Then for every $\alpha \in \text{GEN}(T_2)$ we have $\text{GEN}(T_2)[\alpha] = \text{GEN}(T')[\alpha] = \text{GEN}(T'')[\alpha]$, except for the root labels; but we also have $T_1[\lambda] = T'[\lambda] = T''[\lambda]$, so $T' = T''$. □

Lemma A.3: $(\lhd_I \cup \lhd_S)^* = \lhd$

Proof: Immediate from lemma A.2. □

Lemma A.4: 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 $(x,y) \in \lhd_S \setminus \lhd_M$. Then $x[\lambda] = y[\lambda]$, so no $\lhd_I \setminus id$ steps can take place on a path from x to y. Hence, $(x,y) \in \lhd_M^* = \lhd_M$, which is a contradiction. The result for $\lhd_N$ is proved similarly. □

Lemma A.5: Let P be a partial order, where all closed intervals are finite. Let $P_1, P_2 \subseteq P$ so that $P_1^* = P_2^* = P$. 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 $(x,y) \in id = (P_1 \cap P_2)^0$, 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 $(x,y) \in (P_1 \cap P_2)^*$. □

Lemma A.6: 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 proposition 4.1, all closed intervals of $\mathcal{U}$ are finite, so by lemma A.5, $\lhd = ((\lhd_I \cup \lhd_S) \cap (\lhd_M \cup \lhd_S))^* = ((\lhd_I \cap \lhd_M) \cup \lhd_S)^*$. By lemma A.4, $\lhd_I \cap \lhd_M$ 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. □

Proof of Theorem 4.3: Combine lemmas A.1, A.2, A.3, and A.6. □

Appendix B: Proof of Theorem 4.6

Lemma B.1: If $T_1 \lhd T_2$ then we can define a map $\text{MAP}(T_1,T_2) : \mathcal{U} \rightarrow \mathcal{U}$, which is a $\lhd_G$-monotone, partial function where $\text{MAP}(T_1,T_2)(X) = T_2 \downarrow \alpha$ if $T_1 \downarrow \alpha = X$.

Proof: $\text{MAP}(T_1,T_2)$ is well-defined since $T_1 \downarrow \alpha = T_1 \downarrow \beta = X$ implies $T_2 \downarrow \alpha = T_2 \downarrow \beta$. It is monotone since $T_1 \lhd T_2$ implies $T_1 \lhd_G T_2$ which again implies $T_1 \downarrow \alpha \lhd_G T_2 \downarrow \alpha$. □

Proof of Theorem 4.6: Let $G = \text{GEN}(T)$ and let M be the union of $\text{MAP}(V_j, W_j)$ for which $G(i) = V_j$ for some i. Since $\vec{V}$ and $\vec{W}$ are consistent, M will be a well-defined function.

We construct a new generator $G(\vec{V} \leftarrow \vec{W})$ which defines $T(\vec{V} \leftarrow \vec{W})$. In G we identify the unique set of maximal subtrees that belong to the domain of M. These are necessarily disjoint proper subtrees. Now, $G(\vec{V} \leftarrow \vec{W})$ is obtained from G by substituting every such subtree X by $M(X)$.

We first show $T(\vec{V} \leftarrow \vec{W}) \in \text{SUB}(T, \vec{V}, \vec{W})$. Since M is monotone, it follows that labels in T must be smaller than the corresponding ones in $T(\vec{V} \leftarrow \vec{W})$. Suppose $T \downarrow \alpha = T \downarrow \beta = X$. If X is in the domain of M, then $T(\vec{V} \leftarrow \vec{W}) \downarrow \alpha = T(\vec{V} \leftarrow \vec{W}) \downarrow \beta = M(X)$; otherwise, $T(\vec{V} \leftarrow \vec{W}) \downarrow \alpha = T(\vec{V} \leftarrow \vec{W}) \downarrow \beta = X$. Hence, stability holds, too. The root label is unchanged, so $T \lhd_S T(\vec{V} \leftarrow \vec{W})$. Finally, if $T(i) = V_j$ then $T(\vec{V} \leftarrow \vec{W})(i) = M(V_j) = W_j$.

Suppose that also $T' \in \text{SUB}(T, \vec{V}, \vec{W})$. We show that all labels in $T(\vec{V} \leftarrow \vec{W})$ are smaller than the corresponding ones in T'. If $T \downarrow \alpha$ is in the domain of M, then $T(\vec{V} \leftarrow \vec{W})[\alpha] = T'[\alpha]$. For all other labels, $T(\vec{V} \leftarrow \vec{W})[\alpha] = T[\alpha] \leq T'[\alpha]$, since $T \lhd_S T'$. □

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

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

[8] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Proc. OOPSLA'89, ACM SIGPLAN Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, 1989.

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

[11] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(1), 1983.

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

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

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

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

[16] B. B. Kristensen, O. L. Madsen, B. MøIler-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.

[17] 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.* ACM, 1989.

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

[19] Ole L. Madsen and Birger MøIler-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.

[20] Ole Lehrmann Madsen, Boris Magnusson, and Birger MøIler-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.

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

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

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

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

[25] 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.

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

[27] 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*. Sigplan Notices, 21(11), November 1986.

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

[29] Michael I. Schwartzbach and Erik M. Schmidt. Types and automata. In preparation, 1990.

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

[31] P. Wegner. Dimensions of object-based language design. In *Proc. OOPSLA'87, Object-Oriented Programming Systems, Languages and Applications*, 1987.

[32] P. Wegner and S. 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*. Springer-Verlag (*LNCS* 322), 1988.