What is Type-Safe Code Reuse?

Jens Palsberg
palsberg@daimi.au.dk

Michael I Schwartzback
mis@daimi.au.dk

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

December 1990

Abstract

Subclassing is reuse of class definitions. It is usually tied to the use of class names, thus relying on the order in which the particular classes in a program are created. This is a burden, however, both when programming and in theoretical studies.

This paper presents a structural notion of subclassing for typed languages. It is a direct abstraction of the SMALLTALK interpreter and the separate compilation technique of MODULA. We argue that it is the most general mechanism which can be supported by the implementation while relying on the type-correctness of superclasses. In short, it captures *type-safe code reuse.*

Keywords: structural subclassing, type-safety, separate compilation.

1 Introduction

Object-oriented programming strives to obtain reusable classes without introducing significant compiling or linking overhead. A statically typed language should thus offer general mechanisms for reusing classes without ever

requiring a compiler to re-type-check an already compiled class. Such mechanisms allow *type-safe code reuse*. Instead of suggesting new mechanisms and then later worry about implementation, we will analyze a particular implementation technique and from it derive the most general mechanism it can support. The result is a structural subclassing mechanism which generalizes inheritance.

In the following section we further motivate the notion of type-safe code reuse and discuss our approach to obtain mechanisms for it. In section 3 we discuss a well-known way of implementing classes and inheritance, and suggest a straightforward, inexpensive extension. In section 4 we show that the way code is reused in the implementation can be abstracted into a general subclass relation which captures type-safe code reuse. Finally, in section 5 we give an example.

2 Motivation

To be useful in practice, an object-oriented language should be statically typed and allow separate compilation of classes. The languages C++ [12] and Eiffel [7] come close to achieving this, though the type systems of both have well-known loopholes. Similar to Modula [13] implementations, a compiler for these languages needs only some symbol table information about previously compiled classes. In particular, this is true of the superclass of the class being compiled. Hence, the implementation of a subclass both reuses the *code* of its superclass and relies on the *type* correctness of the corresponding source code. We call this *type-safe code reuse*.

In the following we discuss our approach to type-safe code reuse, the concept of structural subclassing, and a novel idea of class lookup.

2.1 Our approach

From a purist's point of view, the loopholes in the C++ and Eiffel type systems are unacceptable. In search for improvements, one can attempt to alter one or more of the subclassing mechanism, the type system, and the compilation technique. Previous research tends to suggest new type systems

for languages with inheritance, but to ignore compilation.

This paper takes a radically different approach: We analyze the SMALLTALK [4] interpreter together with a well-known technique for separate compilation of MODULA modules, extend them, and derive a general subclassing mechanism for type-safe code reuse. This subclassing mechanism turns out to be exactly the one which we earlier have shown to be spanned by inheritance and type substitution (a new genericity mechanism) [9, 8]. Our analysis of the compilation technique is based on the strong assumptions that types are classes and that variables can only contain instances of the declared class. In [10] we demonstrate that one can replace the type system by a more general one where types are (possibly infinite) sets of classes and subtyping is set inclusion, while retaining the general subclassing mechanism and the compilation technique.

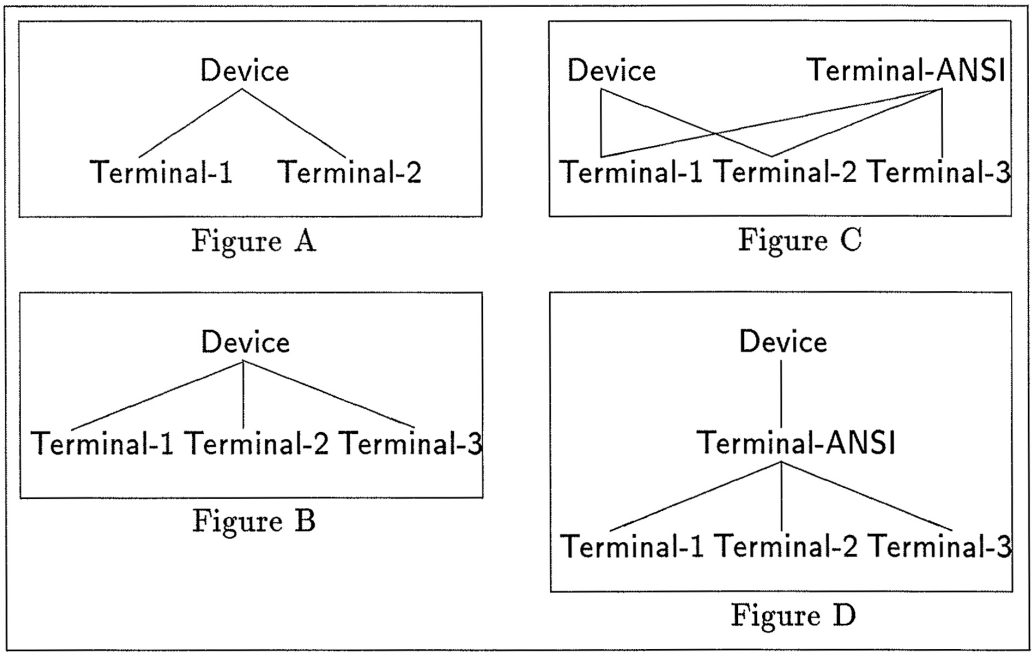

Figure 1: Hierarchies of terminals.

2.2 Structural subclassing

Subclassing is usually tied to the use of class names. This means that a class is a subclass of only its ancestors in the explicitly created class hierarchy. In other words, a superclass must be created *before* the subclass. For an example, see figure 1A where Device must be created before Terminal-1 and Terminal-2.

Suppose that a new type of terminal, Terminal-3, is going to be implemented. An obvious possibility is to implement it as a subclass of Device, see figure 1B. Pedersen [11] discusses the case where the programmer realizes that all three terminals actually are ANSI terminals, i.e., they support the ANSI-defined control sequences. He argues the need for a new mechanism, *generalization*, which would allow the creation of a common superclass, Terminal-ANSI, which should contain all commonalities of the two existing classes. The programmer can then write Terminal-3 as a subclass of Terminal-ANSI, see figure 1C. This is of course not possible when only inheritance (tied to class names) is available, because it forces the class hierarchy to be constructed in a strictly top-down fashion.

Although the mechanism of generalization provides extra flexibility, it does not allow us to create Terminal-ANSI as *both* a common superclass of the three terminals *and* a subclass of Device, see figure 1D. We could of course restructure the class hierarchy by hand, but this may be undesirable or even practically impossible. Our conclusion is that tying subclassing (and generalization) to class names is too restrictive in practice. If subclassing was *structural*, then Terminal-ANSI could be created using inheritance or generalization, or it could even be written from scratch; the compiler will in any case infer the relationship in figure 1D.

Also in theoretical studies a structural notion of subclassing would be preferable. The point is that if all classes and subclass relations are given *a priori*—independently of the programmer's definitions—then they are easier to deal with mathematically. This idea lies behind almost all theories which study types independently of particular programs, see for example [1, 2].

The result of describing a structural subclassing mechanism from existing implementation techniques is a sound basis for theoretical investigations of subclassing and subtyping in ob ject-oriented programming.

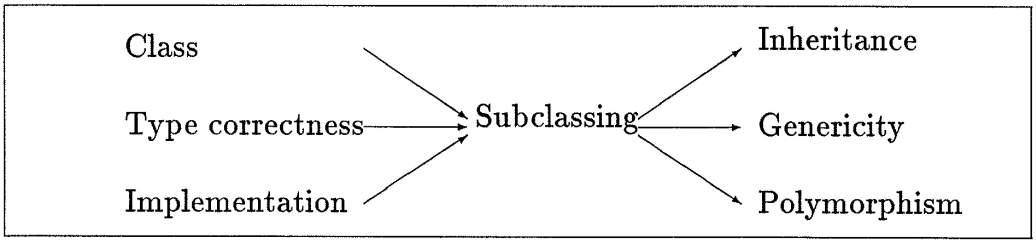

Figure 2: A development of ideas.

We have already reported some of these investigations in other papers [9, 8, 10], see the overview in figure 2. Originally, we simply *defined* the subclassing mechanism that we have now derived. It turned out to have many nice mathematical properties and it lead us to discover a new genericity mechanism (type substitution) which is a significant improvement compared to parameterized classes. It also provided an appropriate setting for analyzing polymorphism and subtyping, leading to a unified type system for object-oriented programming. All these results are now based on the well-understood concepts of class, type correctness, and implementation—rather than some random looking definition of subclassing.

We consider a core language for object-oriented programming with objects, classes, instance variables, and methods. Possible source code in methods include assignments, message passing, and the new expression for creating instances of a class.

Our basic idea is that if the implementation of a class *can* be reused in the implementation of another class, then the two classes should be subclass related. This yields a structural notion of subclassing; it is not tied to the use of class names.

2.3 Class lookup

Our extension of the standard implementation technique is based on the observation that just as reimplementation of methods can be implemented by dynamic method lookup, then redefinition of the arguments of new expressions can be implemented by an analogous *class lookup*. This requires, in a naive implementation, an entry on run-time for each class occurring in a new

expression. Our reason for introducing this extra flexibility is the following. When an instance of for example a list class is created by a method of the list class itself, see figure 3, then the occurrence of list in new list is a recursive one [3].

```
class list
  ... new list ...
end list
```

Figure 3: A recursive list class.

In EIFFEL, this recurrence can be made explicit by writing like Current instead of list. Analogously, in SMALLTALK, one can write self class. Now in a subclass of list, say recordlist, what kind of instance should be created? Meyer [7] argues that the programmer in some cases wants an instance of list and in others an instance of recordlist. In EIFFEL, a statement corresponding to new list would cause the creation of the former, and new (like Current) the latter. With our technique, an instance of recordlist will always be created—the choice that will most often be appropriate. The generality of EIFFEL can be recovered, however, using *opaque* definitions [9], but this will not concern us here. This means that in recordlist the recursive occurrence of list is implicitly substituted by recordlist. But why, we ask, should only the class in *some* but not all new expressions be substitutable? By introducing class lookup, we remove this unpleasing asymmetry. The notion of *virtual class* in BETA [5, 6] is actually implemented by a variation of class lookup.

Let us now move on to a description of how to implement classes, inheritance, and instance creation.

3 Code Reuse

We will describe interpreters for three languages of increasing complexity. The first involves only classes and objects, and its implementation is essentially that of separately compiled modules in MODULA. The second language introduces inheritance which is implemented as in the SMALLTALK interpreter, except that we retain separate compilation. The third language

extends this with the possibility of redefining the arguments of new expressions. This is implemented using class lookup which is analogous to method lookup. Throughout, we focus solely on those concepts that have impact on the structural subclassing mechanism which we derive in a later section.

3.1 Classes

Classes group together declarations of variables and methods. An *instance* of a class is created by allocating space for the variables; the code for the methods is only generated once. The compiler uses a standard symbol table containing names and types of variables and procedure headers. On runtime three structures are present: The *code space* which implements all the methods, the *object memory* which contains the objects, and the *stack* which contains activation records for active methods. An *object* is a record of instance variables, each of which contains either nil or a pointer to an object in the object memory. The situation is illustrated in figure 4.

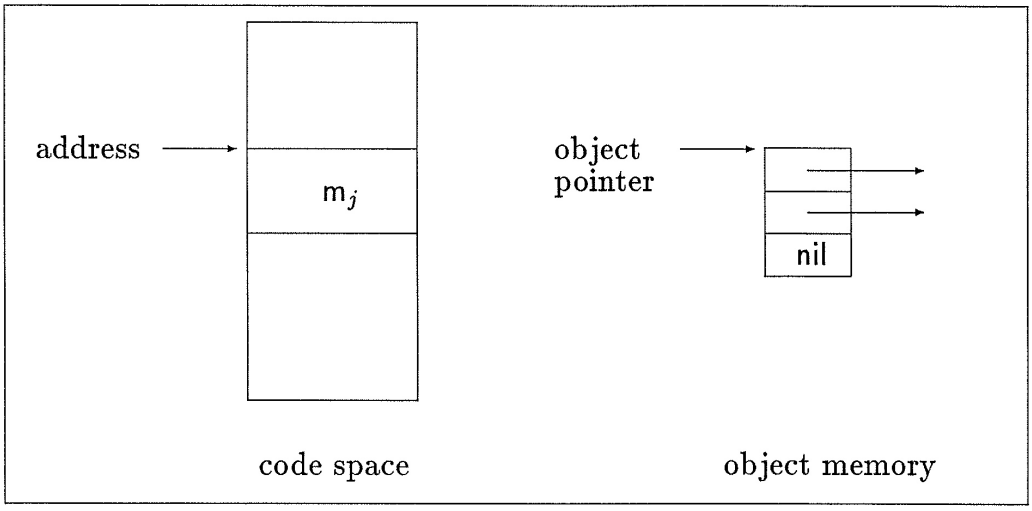

Figure 4: Implementation of classes and objects.

To present the workings of the interpreter, we shall sketch the code to be executed for two language constructs: message sends and object creations. A message send of the form $x.m(a_1, \ldots, a_k)$ generates the following code:

```
PUSH a_1
⋮
PUSH a_k
CALL ADDRESS(m)
```

Notice that the compiler can statically determine the address of the method, since the class of the receiver x is known. The code for the object creation new C is:

```
ALLOCATE(nil, ..., nil)
```

with one argument for each instance variable in C. This operation returns an object pointer to a record with fields initialized by the arguments. Again, the number of instance variables is statically known by the compiler.

3.2 Inheritance

The concept of inheritance allows the construction of subclasses by adding variables and methods, and by replacing method bodies. On run-time is introduced an important new structure: the *class table*, which for each class C describes its superclass, its number of instance variables, and its *method dictionary* associating code addresses to method names. At the same time an object record is extended to contain the name of its class (in the form of a class pointer). The situation is illustrated in figure 5. Also the symbol table is slightly changed. Analogously to how the class table is organized, all entries for classes contain the name of its superclass.

The activation record for a message send will now contain the receiver; it can be thought of as an implicit actual parameter and will be accessible through the metavariable SELF. The code for a message send is now:

```
PUSH (x)
PUSH a_1
⋮
PUSH a_k
CALL M-LOOKUP(CLASS(x), m)
```

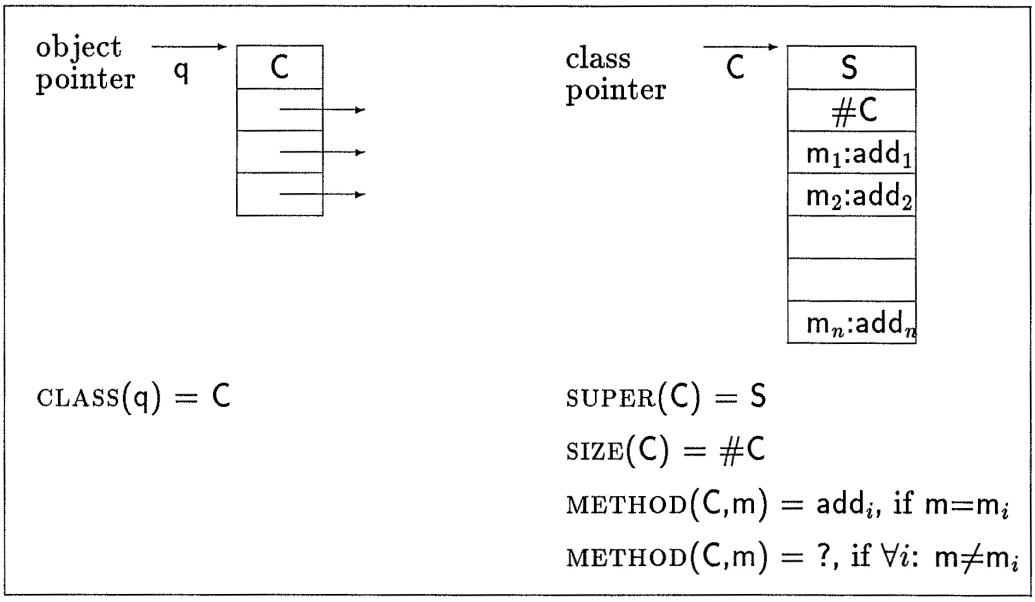

Figure 5: Implementation with inheritance.

where the *method lookup* is definde as follows

$$\text{M-LOOKUP}(\mathsf{q},\mathsf{m}) = \begin{cases} \mathsf{message\text{-}not\text{-}understood} & \text{if } \mathsf{q} = \mathsf{nil} \\ \mathsf{add} & \text{if } \text{METHOD}(\mathsf{q},\ \mathsf{m}) = \mathsf{add} \neq ? \\ \text{M-LOOKUP}(\text{SUPER}(\mathsf{q}), \mathsf{m}) & \text{otherwise} \end{cases}$$

The code for object creation comes in two varieties. For non-recursive occurrences, such as new C, we generate the code:

$$\text{ALLOCATE}(\mathsf{C},\ \mathsf{nil},\ \ldots,\ \mathsf{nil})$$

which just includes the class in the object record. For rexursive occurrences, we must generate the code:

$$\text{ALLOCATE}(\text{CLASS}(\text{SELF}),\ \mathsf{nil},\ \ldots,\ \mathsf{nil})$$

with (CLASS(SELF)) nil-arguments.

3.3 Object creation

We now depart from the standard interpreters by allowing a subclass to modify the classes that are used for object creation. For each occurrence a of a new expression we introduce an *instantiator*. The class description now contains an *instantiator dictionary* association classes to the instantiators. Finally, we introduce *instantiator lookup* analogously to method lookup. The situation is illustrated in figure 6.

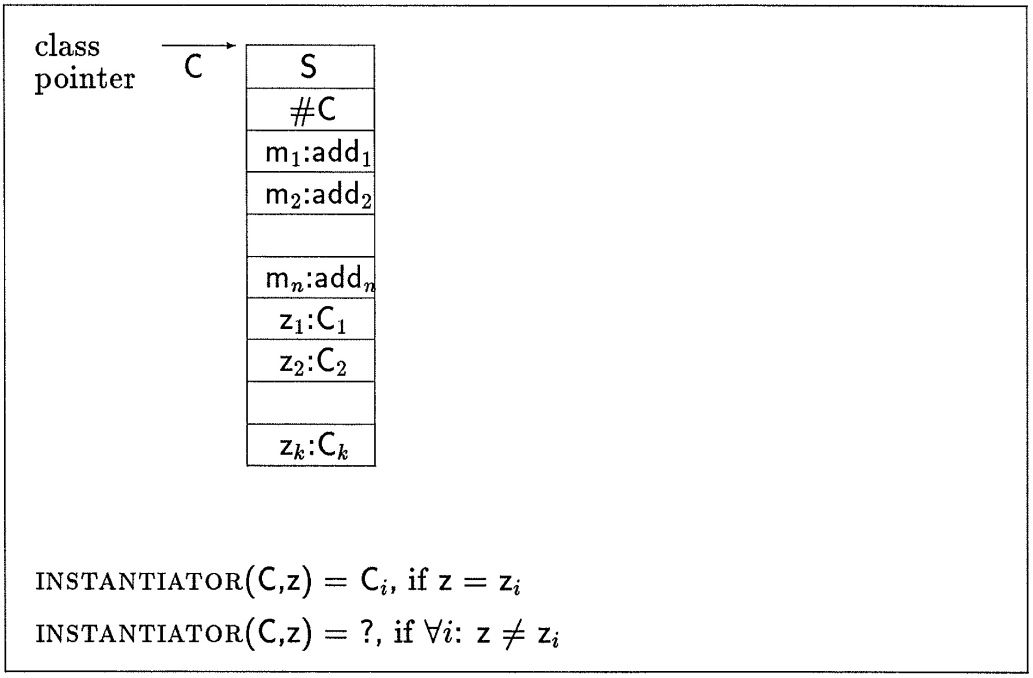

Figure 6: Implementation with inheritance and instantiators.

The code for a nonrecursive object creation, such as new z where z is now an instantiator, is

$$\begin{array}{l} \mathsf{C} \leftarrow I\text{-}\textsc{lookup}(\textsc{class}(\textsc{self}))),\mathsf{z}) \\ \textsc{allocate}(\mathsf{C},\ \mathsf{nil},\ \ldots,\ \mathsf{nil}) \end{array}$$

with SIZE C nil-arguments. The instantiator lookup is defined as follows:

$$\text{I-LOOKUP}(\mathsf{q},\mathsf{z}) = \begin{cases} \mathsf{instantiator\text{-}not\text{-}found} & \text{if } \mathsf{q} = \mathsf{nil} \\ \mathsf{C} & \text{if } \text{INSTANTIATOR}(\mathsf{q},\, \mathsf{z}) = \mathsf{C} \neq ? \\ \text{I-LOOKUP}(\text{SUPER}(\mathsf{q}), \mathsf{z}) & \text{otherwise} \end{cases}$$

The code for recursive occurrences is the same as before.

4 Type-Safe Code Reuse

The separate compilation of a class C yields an extension of the symbol table, the class table, and the code space. A triple such as

(symbol table, class table, code space)

we will call a *context* and usually denote by the symbol Θ. Thus, we can view the compilation process as a mapping from contexts to contexts. Notice that since we want the usual notion of separate compilation, only the symbol table and the class table can be inspected during the compilation of a new class.

We assume that the compilation ensures that the class is type-correct, which includes the usual syntactic checks as well as *early* checks and *equality* checks. For every message send of the form x.m(...) an early check requires that a method m with the appropriate number of parameters is implemented in the declared type of x. For every assignment of the form x:=y, and similarly for parameter passings, an equality check requires that the declared types of x and y are equal.

A context completely describes an implementation of a collection of classes. Obviously, there are many ways of achieving the same result, depending on how the class hierarchy is organized. Thus, we can introduce a notion of *equivalence* of contexts, $\Theta_1 \approx \Theta_2$, whenever the two respond alike to every request of the form SIZE, M-LOOKUP, and I-LOOKUP described in the previous section. The difference between two equivalent contexts is the degree to which the possibilities for code reuse have been exploited.

These possibilities can be expressed in terms of *extensions* of contexts. If C is a class defined in a context Θ, then a C-extension of Θ is just the information required to construct a new subclass in the class hierarchy. Hence, it is again

a triple consisting of a symbol table, a class table, and a code space. The only difference between an extension and a context is that not all SUPER-pointers need to be defined in the former, whereas the latter is completely self-contained. This is illustrated further in figure 8 in section 5.

4.1 General Subclassing

A class C_2 is said to be a Θ-*subclass* of the class C_1 when they are both defined in Θ and C_1 occurs in the SUPER-chain of C_2. This is a concrete notion of subclassing. We can give another notion which captures the *potential* subclass relations.

Consider the source code of two classes C_1 and C_2. Whether they are in a subclass relation or not depends on Θ, as follows.

$\mathsf{C}_1 \lhd_\Theta \mathsf{C}_2$

$\Updownarrow$

$\exists$ a C_1-extension E such that the result of
- compiling C_1 in Θ and then extending with E; and
- compiling C_2 in θ

are equivalent

This definition clearly expresses that C_2 *could* be implemented as a subclass of C_1. It is not the full story, however.

Because of our adherence to separate compilation, the extension above should be insensitive to changes in the implementation of methods in C_1.

Let us call C and E *compatible* in Θ if compiling C in Θ and extending with E is equivalent to the compilation of some other class in Θ. In the definition of a $\lhd_\Theta$ above we will only allow extensions that are compatible with *all* classes that have the same symbol table as C_1.

This definition is not quite what we want, since it relies very heavily on details of the implementation and programmer-defined names. We want a more abstract, *structural* notion of subclassing. To be able to define a such, let us introduce a slight abstraction of the source code of a class.

4.2 Classes as Trees

We shall represent a class as an ordered, node-labeled tree. Given a class name C and a context Θ, we can reconstruct the untyped code of its implementation, by short-circuiting the SUPER-chains and collecting all relevant information. This code will be the label associated with the root of the tree. In place of each occurrence of a class name we supply the tree corresponding to that class. This will in general yield an infinite tree, due to recursion; however, since Θ is always finite, the tree will be *regular*, i.e., it will only have finitely many *different* subtrees. We shall denote this tree by $\textsc{tree}_\Theta(\mathsf{C})$.

We can now lift the subclass relation to trees, as follows.

$$\begin{array}{ll} & \mathsf{T}_1 \lhd \mathsf{T}_2 \\ \Updownarrow & \\ & \exists\, \Theta, \mathsf{C}_1, \mathsf{C}_2\colon \quad \mathsf{C}_1 \lhd_\Theta \mathsf{C}_2 \wedge \\ & \qquad\qquad \textsc{tree}_\Theta(\mathsf{C}_1) = \mathsf{T}_1 \wedge\ \textsc{tree}_\Theta(\mathsf{C}_2) = \mathsf{T}_2 \end{array}$$

It follows directly from this definition that

$$\begin{array}{ll} & \mathsf{C}_1 \lhd_\Theta \mathsf{C}_2 \\ \Updownarrow & \\ & \textsc{tree}_\Theta(\mathsf{C}_1) \lhd\ \textsc{tree}_\Theta(\mathsf{C}_2) \end{array}$$

Note also that we now have a structural equivalence on classes defined as equality of the corresponding trees with respect to some Θ.

4.3 Structural Subclassing

Although the notion of a class has been made more abstract, by the representation as a tree, the subclass relation is still explicit about contexts. However, we can phrase the $\lhd$ relation in a pure tree terminology. Let us call the above definition of $\lhd$ for $\lhd_{\text{IMPL}}$, since it relates directly to the implementation. The alternative definition will be called $\lhd_{\text{TREE}}$ and is given below.

We first need to define the notion of the *generator* of a tree. It is obtained by replacing all maximal recursive occurrences of the tree in itself by the special label $\diamond$. If T is a tree, then $\textsc{gen}(\mathsf{T})$ is its generator—another tree.

We also need a bit of notation. A *tree address* is simply an indication of a path from the root to a subtree. We shall write $\alpha \in \mathsf{T}$ when α is a valid tree address in T. In that case $\mathsf{T} \downarrow \alpha$ denotes the corresponding subtree, and $\mathsf{T}[\alpha]$ denotes the label in the root of that subtree.

We can now define

$$\begin{array}{l} \mathsf{T}_1 \lhd_{\text{TREE}} \mathsf{T}_2 \\ \Updownarrow \\ \forall \alpha \in \mathsf{T}_1 : \textsc{gen}(\mathsf{T}_1 \downarrow \alpha) \lhd_G \textsc{gen}(\mathsf{T}_1 \downarrow \alpha) \end{array}$$

where $\lhd_G$ is defined by

$$\begin{array}{ll} \mathsf{G}_1 \lhd_G \mathsf{G}_2 & \\ \Updownarrow & \\ \text{Monotonicity:} & \forall \alpha \in \mathsf{G}_1 : \mathsf{G}_1[\alpha] \leq \mathsf{G}_2[\alpha] \\ \text{Stability:} & \forall \alpha, \beta \in \mathsf{G}_1 : \mathsf{G}_1 \downarrow \alpha = \mathsf{G}_1 \downarrow \beta \Rightarrow \mathsf{G}_2 \downarrow \alpha = \mathsf{G}_2 \downarrow \beta \end{array}$$

The essence of $\lhd_G$ is that code can only be extended, and equal classes must remain equal.

This definition contains no mention of implementations; nevertheless, we can show that $\lhd_{\text{IMPL}} = \lhd_{\text{TREE}}$. Thus, it does satisfy our requirements of being an independent, structural subclass relation that is at the same time deeply rooted in implementation practices. The above definition of $\lhd$ is the basis of the papers [9, 8, 10].

4.4 Formalities

We can sketch a demonstration of the above equality in the form of two inclusions. **The inclusion** $\lhd_{\text{TREE}} \subseteq \lhd_{\text{IMPL}}$:
Assume that $\mathsf{T}_1 \lhd_{\text{TREE}} \mathsf{T}_2$. We must construct $\Theta, \mathsf{C}_1, \mathsf{C}_2$ with the appropriate properties. We shall in fact provide an inductive method for doing this. The induction will proceed in the number of different subtrees in the T_i's; this is

a finite number since the trees are regular. If they have no subtrees, then their implementation is trivial. Otherwise, we first compute the generators of the two trees, i.e., we discount their recursive occurrences. The remaining subtrees form a strictly smaller set, since two trees have been removed. To every remaining immediate subtree of T_1 there is a corresponding $\lhd_{\text{TREE}}$-related immediate subtree of T_2. By induction hypothesis, we can implement all of these subtrees in some context. The extraneous subtrees of T_2 can be trivially implemented leading to the final, larger context Θ. We must now show how to extend this to T_1 and T_2. The code C_i for T_i is essentially a named version of the root label, with class names from Θ in place of subtrees, and the name of the code itself in place of the recursive $\diamond$-occurrences. It should be clear that $\lhd_{\text{TREE}}(\mathsf{C}_i) = \mathsf{T}_i$. The extension of C_1 that will be equivalent to C_2 is clearly a class table with C_1 as SUPER, with SIZE equal to the number of instance variables in C_2, with a method dictionary reflecting the extra code, and with an instantiator dictionary reflecting the substitution of classes from C_1 to C_2. From the previous construction we see that the instantiator dictionary only substitutes Θ-subclasses. From monotonicity and stability it follows that this extension is compatible with all modifications of C_1 that do not change the symbol table. The result follows.

The inclusion $\lhd_{\text{IMPL}} \subseteq \lhd_{\text{TREE}}$:
We now look at the situation where $\mathsf{C}_1 \lhd_\Theta \mathsf{C}_2$, for some Θ, C_1, C_2. We must show that the associated trees are $\lhd_{\text{TREE}}$-related. Again, we proceed by induction, this time in the size of the implementation of the C_i's. We will automatically have monotonicity of the root labels. Hence, if neither C_i refers to other classes, then their associated trees are trivially $\lhd_{\text{TREE}}$-related. Otherwise, we consider any class name N_1 mentioned in C_1, which is not a recursive occurrence, and the corresponding class name N_2 mentioned in C_2. By construction, N_2 is a Θ-subclass of N_1; hence, by induction hypothesis we conclude that their associated trees are $\lhd_{\text{TREE}}$ related. Now, consider the generators of the trees associated with C_1 and C_2. Monotonicity will always hold, since we can only extend, but we must establish that stability will necessarily hold. This comes from the requirement that the extension must be compatible with all modifications of C_1 that leave the symbol table unchanged: Assume that that two tree addresses indicate the same class in the tree of C_1. If this is a recursive occurrence, then the two classes in C_2 will automatically also be equal, since they will all refer to CLASS(SELF). Otherwise, we claim that the instantiator dictionary in the class table of

the extension must let the two classes remain equal. If not, then we could construct a modification of C_1 in which the two classes were types of variables that were the arguments of a legal assignment. In C_2 this assignment would become illegal, so the extension would not be compatible with the modification. The result follows.

5 Example

To illustrate the construction of the structural subclassing mechanism, we examine an example program, see figure 7.

```
class C
  var x: integer
  method p(arg: boolean)
    ... new object
    ... new C ...
end C
class D
  var x: integer
  method p(arg: boolean)
    ... new boolean
    ... new D ...
  var y: integer
  method q
    ...
end D
```

Figure 7: An example program.

Class D differs from C in having other arguments to the two occurrences of new and in declaring an extra variable and an extra method. This can be made explicit in the implementation by using C as the super part in the entries for D and then only specifying the differences from C, see figure 8.

This includes specifying that one of the instantiators is new boolean. It does *not*, however, include any specification of the replacement of new C by new D. This is because the occurrence of C is a recursive one; hence, the code for

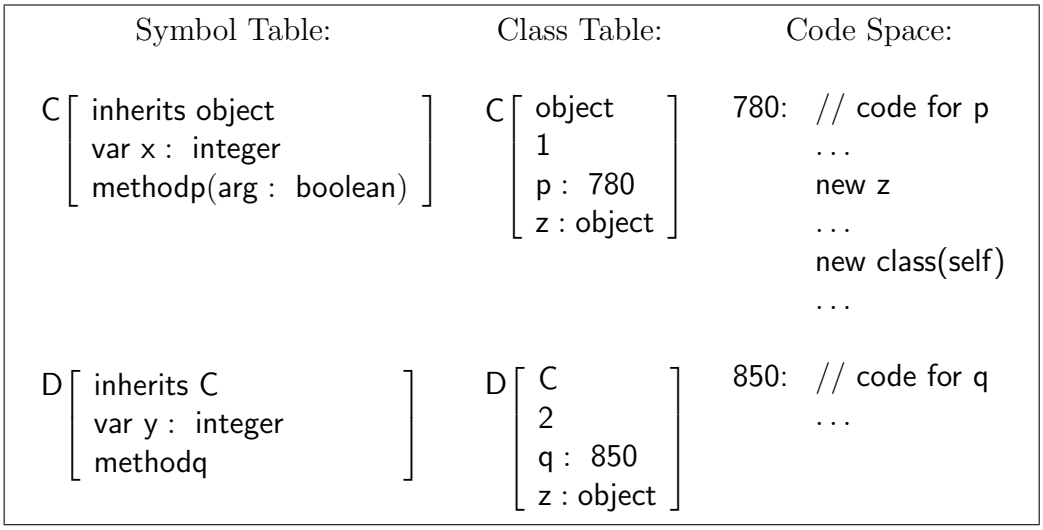

Figure 8: An implementation of the example program.

it is new class(self). Likewise, the occurrence of D is recursive; thus, we use the same code as before.

The incremental implementation shows that C and D should be subclass related. Indeed, we can specify D as explicitly being a subclass of C by using the standard syntax for inheritance together with the syntax for type substitution that we introduced in [9], see figure 9.

```
class D inherits C[object ← boolean]
  var y: integer
  method q
    ...
end D
```

Figure 9: Class D as an explicit subclass of C.

The generators for C and D are given in figure 10. They can be obtained from the implementation as follows. Consider first class C. In the symbol table we find that its super entry contains object so this yields the empty code sequence. Next, we extend (!) that by introducing the declarations listed in the symbol table entry for C. Note that occurrences of class names become subtrees. Finally, we extend with the code for p using class table

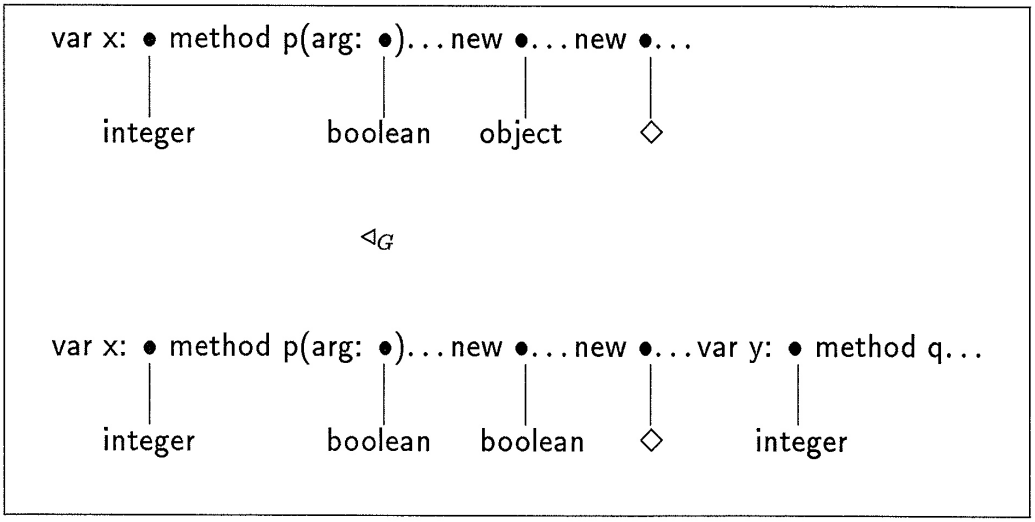

Figure 10: The generators for the example classes.

information about instantiators. Occurrences of new class(self) become new together with a $\diamond$-subtree.

Next, consider class D. Its super entry contains C, so this yields the tree from before. Using the symbol table entry for D we can then extend it by introducing two more declarations. The only pitfall is the instantiator z in the class table. It makes us find the occurrence of z in the code, find the corresponding place in the tree, and substitute the subtree object by boolean.

To see that the two constructed generators are $\lhd_G$-related, notice that subtrees only get larger, and that stability trivially holds because no two subtrees in the generator for C are equal. Hence, the trees corresponding to the classes C and D are $\lhd$-related.

6 Conclusion

We have analyzed a particular implementation technique for typed object-oriented languages, which allows separate compilation and generalizes the usual SMALLTALK interpreter. From this we obtained the relation $\lhd_\Theta$ which captured the maximal *potential* for type-safe code reuse. Finally, we ab-

stracted classes into trees, and defined $\lhd$, which is an equivalent, structural version of $\lhd_\Theta$. Structural subclassing provides more flexibility than subclassing tied to class names; it is also an appropriate basis for theoretical studies.

The implementation we have described cannot immediately cope with *mutually* recursive classes, but it can fairly easily be extended to deal with this complication—at a small cost on run-time. The theory of trees and $\lhd$ can, however, handle such an extension without any changes at all. This is because $\diamond$ can occur in any leaf and not just immediately below the root.

References

[1] 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.

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

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

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

[5] 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.

[6] 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.

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

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

[9] 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.

[10] Jens Palsberg and Michael I. Schwartzbach. A unified type system for object-oriented programming. Computer Science Department, Aarhus University. Submitted for publication, 1990.

[11] 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,* 1989.

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

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