Method reuse in typed object-oriented languages*

Ryan Stansifer
University of North Texas
Department of Computer Sciences
University of North Texas
Denton, Texas 76203–3886
Internet: `stansifer@cs.unt.edu`
Telephone: (817) 565–2811

Dan Wetklow
University of Pittsburgh at Johnstown
Department of Computer Science
University of Pittsburgh at Johnstown
Johnstown, PA 15956
Telephone: (814) 269–2926

October 11, 1994

Abstract

In some object-oriented languages it is possible to inherit selected methods from superclasses. We refer to this as selective inheritance. The obvious approach to selective inheritance in previous record-based models of classes requires unnecessary type restrictions. In this paper we propose a slightly different model which is more lenient in this regard. In addition to its advantages when selective inheritance is used, this model is also more permissive than the usual models with respect to the typing of class expressions where methods are overridden. Furthermore, our approach allows more classes to be typed in cases where methods are overridden. Thereby, the need

*An preliminary version of this paper appeared in the Symposium on Applied Computing.

for complex typing mechanisms are eliminated in some cases. We discuss modeling object-oriented languages and the implications of our proposed model.

1 Introduction

One way programming languages promote the reuse of code is through inheritance [12, 20]. This paper is about maximizing code reuse in statically-typed object-oriented languages. We are interested in statically-typed languages for all the usual reasons: natural organization of data, the early detection of errors, and the efficient execution of programs. Good discussions about the benefits of types in programming languages can be found in several places [2, 4]. A major advantage of a strongly typed object-oriented language is the elimination of the "message not understood" error when an object receives a request that it is not prepared for.

Of particular concern in this paper are the typing rules with respect to inheritance, particularly with respect to multiple inheritance. To illustrate our points we use a simple object-oriented language, and to examine the compile-time properties of inheritance in this language we take the usual approach of defining the object-oriented constructs in terms of lambda expressions. We refer to such a translation as a *model*.

A model permits us to view object-oriented constructs as macro definitions. Thus, typing rules for object-oriented constructs can be directly obtained from the typing rules of the typed lambda calculus and the "macros" defined by the model. Different models will lead to different typing rules for the object-oriented constructs and some typing rules are more restrictive than others. Of primary concern in this paper is the amount of method reuse permitted by different typing rules. Our emphasis is on defining a model that maximizes flexibility, i.e., one that allows the programmer to reuse methods via inheritance while minimizing the type restrictions placed upon this process.

As our goal is to maximize the flexibility that the programmer has with respect to inheritance, we introduce the notion of what we refer to as *selective inheritance*. Essentially, selective inheritance allows one to inherit specified methods from a superclass and, by default, ignore the other methods in that superclass. The primary advantage of selective inheritance is that the methods that are not inherited do not affect the type of the subclass. Since selective inheritance is most useful when multiple inheritance is permitted, the simple object-oriented language used in this paper permits a class to inherit methods from a number of different classes.

Because the previous models [8, 14] were designed for languages that were restricted to single inheritance, there was no mechanism for the selective inheritance of methods in the object-oriented language they modeled. Selective inheritance can be added to the usual models in a straightforward fashion, but we found that unexpected (from the programmer's point of view) type restrictions result. Methods that are not inherited and thus are not present in a class can still affect the type of the class and render it uninstantiable. The model presented in this paper does not permit uninherited or overridden methods to affect the type of a class. Therefore, the situation where a class cannot be instantiated due to type restrictions imposed by methods that are not present in the class is avoided.

An important part of all these models of object-oriented languages is the data type record. The record-based model of object-oriented languages introduced by Cardelli [3] provides a framework for strongly typed object-oriented languages. This model has played a role in the development of Modula-3 [13] and is similar to the approach taken in C++ [9]. All of the models discussed here are refinements of the record-based model of Cardelli.

2 Modeling Object-Oriented Languages

Among the distinguishing features of object-oriented languages is that data values are organized into a hierarchy of *classes*. Classes are templates for creating objects, the concrete instances of a class, and thus they define the general structure of data in the program. Classes may have a suite of *methods* associated with them. In the parlance of object-oriented languages, objects communicate with each other through *messages*, which are requests that one of an object's methods be executed. This is normally the only way objects may communicate with one another, thus there is a similarity between classes and abstract data types.

In this section, the syntax for various object-oriented constructs of a hypothetical language is given, and these constructs are explained informally. In section 3 we make the semantics of these constructs precise by translating them to typed lambda-expressions.

2.1 Object-Oriented Languages

To capture the notion of a class we introduce a construct with the following syntax:

class (v_1, ..., v_n)
methods
  m_1=e_1, ..., m_k=e_k
end

A class with this definition has instance variables $v_1, \ldots, v_n$. The number of instance variables n may be 0. The instance variables may appear in the method definitions $e_1, \ldots, e_k$. (In this paper $k > 0$, but this plays no role.) The names of these methods are $m_1, \ldots, m_k$, respectively.

Since classes are templates for creating objects, some mechanism for creating objects from a class must exist. The syntax:

`new` (C, e_1, ..., e_n)

will denote this mechanism, where C is a class and $e_1, \ldots, e_n$ are expressions supplying values for the instance variables of the class C. In this paper we are not concerned with mutable instance variables or the issue of state, so the values of instance variables are constant.

Methods of an object O are accessed by $O.m$ where m is one of O's methods. Static typing ensures that correctly typed expressions of the language access only the methods that are in the domain of the objects for the same reason that the expression $rec.f$ results in a compile-time type error if there is no field f in the record rec in most statically typed languages.

In object-oriented languages methods may refer to the object itself through the special identifier `self`. Thus the identifier `self` may appear in any of the method definitions $e_1, \ldots, e_k$. By using `self` a method may invoke other methods belonging to the same object.

2.1.1 Inheritance

Another feature of most object-oriented languages is *inheritance*. For our purposes, inheritance is the ability of a subclass to obtain some of the properties of its superclasses. We permit two distinct forms of inheritance: the form that Wegner [20, page 509] calls *class inheritance* and the form that we refer to as *selective inheritance*.

First, we discuss the more common form of inheritance: class inheritance. As an example, let C be the class

`class` (v) `inherits all from` S (e_1) `methods` m=e_2 `end`

where S is also a class (the superclass of C). The instance variable of S is set to e_1, and all of the methods of S are inherited by C. In class C, the method m is defined by the expression e_2. Any occurrence of *self* in an inherited method refers to the instance of C, not to any instance of S.

If a method m is defined by a class C and is also inherited from a superclass of C, then the definition of m in C has precedence over the inherited definition of m. In other words, the version of m that has been inherited is *overridden* by the m that is defined in C.

The `inherits all` construct passes all the methods to the subclass. We also provide a related construct to allow the subclass to inherit only certain named methods. The syntax of *selective inheritance* by a class is as follows:

> `class` (v)
> `inherits` m_1`,` m_2 `from` S (e_1) `methods` m=e_2 `end`

Selective inheritance allows a class to inherit some, but not necessarily all, methods of a superclass. Thus, we have two different mechanisms for inheritance: one that passes on all the methods of the superclass, and one that transfers only the specified methods of the superclass.

2.2 Discussion of Selective Inheritance

In the object-oriented language that we are modeling, more than one `inherits` clause is permitted. Therefore, a class can have multiple superclasses. This is called *multiple inheritance*. In the case of multiple inheritance there are advantages in allowing the selective inheritance of methods. One advantage is that selective inheritance provides a simple mechanism for a subclass to specify from which superclass a method is to be inherited when there is a method name conflict; there is no need for a complicated set of precedence rules.

If the object-oriented language is strongly typed, then there is another advantage of allowing the selective inheritance of methods. It has been shown that multiple `inherits all` clauses cause problems with type reconstruction [17, 18, 21]. In particular, the *principal type* property (all derivable types of an expression can be obtained from a single type in the type system) is lost—the type of an expression must be represented with a set of types. For example, in the expression

> $\lambda P.\lambda Q$`. new (class () inherits all from P inherits all from Q`
> `end).m+3`

class `P` could have had a method `m` of any type which was overridden by the method `m` of class `Q` (which had type integer), or perhaps there was no method `m` in class `P` but class `Q` had a method `m` of type integer, or perhaps class `Q` had no method `m` but class `P` had a method `m` of type integer. It cannot be determined from the code whether method `m` is being inherited from class P or class Q. The range of possibilities cannot be captured with a single type (the principal type) in most type systems. However, if multiple inheritance is obtained by permitting classes to inherit from an arbitrary number of superclasses via selective inheritance and restricting the number of `inherits all` clauses to no more than one, then the difficulties associated with type reconstruction are avoided [21].

2.3 Augmented lambda expressions

Our goal is to translate the object-oriented constructs introduced in the last section into typed lambda expressions augmented with some constructs for records. In this section we make precise the language to which the object-oriented constructs will be translated.

Let m stand for record labels and v for variables. The table below gives the language.

expression	description
v	variable
$e_1(e_2)$	function application
$\lambda v.e$	function definition
μe	fixpoint
$\texttt{<}m_1 = e_2, \ldots, m_k = e_k\texttt{>}$	records
$e.m$	field selection
$e_1 \texttt{\|\|} e_2$	record concatenation
$e_1 \mathbin{@} e_2$	methods application

In addition to the usual constructs of the lambda calculus we have records which are finite associations of values to labels. We denote records using angle brackets, e.g., $\texttt{<}a = 4, b = true\texttt{>}$. Record concatenation, which we denote by $e_1 \texttt{||} e_2$, adds the fields of record e_2 to record e_1. In the case of duplicate labels the the field in e_1 is discarded. For example, the expression $\texttt{<}a = 4, b = v\texttt{>}\texttt{||}\texttt{<}c = 9\texttt{>}$ denotes the same record as $\texttt{<}a = 4, b = v, c = 9\texttt{>}$. Also, $\texttt{<}a = 4, b = v\texttt{>}\texttt{||}\texttt{<}a = true\texttt{>}$ is the record $\texttt{<}a = true, b = v\texttt{>}$. Record concatenation is a different construct from record extension that is found in some other languages [6]. Record extension requires that no field of the second record appear in the first.

It is possible to define records and subtyping in the more primitive terms of the polymorphic lambda-calculus [5]. There is no advantage to doing this here as we are not interested in semantic models of the values, but in the types of the expressions. Records are a common enough data structure that we can add them to the collection of lambda-expressions without confusion.

The last construct in the table is more peculiar and is not used in other models of object-oriented languages. We refer to it as the "methods applicator" operation. We use it as a matter of convenience in the modeling of the `new` construct. The methods applicator takes two arguments: an arbitrary expression e_2 and a record e_1 that has the property that every field in e_1 is a function. The result of this operation is the record obtained by applying every field in e_1 to e_2. For example, the expression $\texttt{<}a = \lambda v.v, b = \lambda v.\lambda x.x, c = \lambda y.\texttt{<}m = y\texttt{>>} \mathbin{@} 3$ has the value $\texttt{<}a = 3, b = \lambda x.x, c = \texttt{<}m = 3\texttt{>>}$.

2.4 Types

Many type systems for the typed lambda calculus augmented with record constructs have been studied [10, 15, 16, 17, 21], and the results in this paper hold for those type systems. Additional type systems are explored in the articles in [1]. In other words, the added flexibility of the model we propose in this paper is not contingent upon a particular type system being used. These relatively complex type systems detract from the purpose of discussing models of inheritance. All we need assume of the type system is:

1. some reasonable collection of types including types for integers, boolean values, records, and function types;

2. the usual typing rules for function application, lambda abstraction, record selection, and so on.

3. some concept of a subtype relation between records (either an explicitly defined subtype relation or a type system which uses parametric polymorphism to simulate a subtype relation).

In typed models of object-oriented languages the subtype relation is important. The type τ_1 is a subtype of the type τ_2 (denoted $\tau_1 \leq \tau_2$) if an expression having type τ_1 can be used anywhere that an expression having type τ_2 can be used without danger of a run-time type error. For this reason it is said that an expression of type τ_1 also has type τ_2.

The value of this form of subtype polymorphism is most evident in the case of records. The record type r_1 is a subtype of the record type r_2 if every field in r_2 is also in r_1. In addition, for every label m in r_2, the relation $r_1.m \leq r_2.m$ holds. The function type $\tau_1 \rightarrow \tau_2$ is a subtype of the function type $\tau_3 \rightarrow \tau_4$ iff $\tau_3 \leq \tau_1$ and $\tau_2 \leq \tau_4$. As an example of the value of subtype polymorphism, consider the situation where a function demands its actual parameter be a record with an `m` field of type `integer`. If the type system permits subtype polymorphism, then any record with an appropriate `m` field can be used as the actual parameter of the function—the fact that the record has additional fields does not result in a type error.

3 Existing Models

In a seminal work by Cardelli [3] ordinary, Pascal-like records were used to account for the basic features of objects. The methods of an object were modeled as record fields, generally with functional types. In this model a class is modeled as a function from instance variables to records. More refined models [8, 14] have been put forth which capture the purpose of the so-called "pseudo" variable[1] *self*. It is these models that we refer to as the "usual models." In the next section we compare our model with them.

3.1 Records

In Cardelli's model, the class construct:

```
C=class (v1, ..., vn)
methods
  m1=e1, ..., mk=ek
end
```

can be modeled as:

$$C = \lambda v_1. \ldots . \lambda v_n. \texttt{<} m_1 = e_1, \ldots, m_k = e_k \texttt{>}$$

which is a function that takes n parameters and returns a record. The result of a function call to C is an object, thus, the *new* construct can be modeled as a function call to C.

3.2 Modeling `self`

The identifier *self* can be used in the methods of an object to refer to other methods in the object, or to the object itself. For example, the methods of some hypothetical class $m_1 = 3$, and $m_2 = \mathit{self}.m_1 + 1$ are computationally equal to $m_1 = 3$, and $m_2 = 3 + 1$.

The key is to model a class as a function from *self* to a record and an object as a fixpoint of this function. For example, the class C

```
class (v1, ..., vn)
methods
  m1=e1, ..., mk=ek
end
```

is modeled as

$$\lambda v_1. \ldots . \lambda v_n. \lambda \mathit{self}. \texttt{<} m_1 = e_1, \ldots, m_k = e_k \texttt{>}$$

The creation of an object from a class is not a simple function application that binds the values of all the instance variables. Rather it requires taking the fixpoint of a class. For

[1] These identifiers are called pseudo variables in the object-oriented parlance first introduced by Smalltalk, because they have special values, unlike "real" variables. Of course, as these models make clear, they are unlike variables because they are actually bound identifiers.

example, $C(e_1, ..., e_n)$ is a function from *self* to a record and the fixpoint of that function, $\mu C(e_1, ..., e_n)$, is an object. In this way *self* is bound to the object as a whole. Whether or not there are any instance variables makes no difference in this approach.

Inheritance is also accounted for in the usual models of object-oriented languages. The construct C

```
class (v) inherits all from S methods m=e end
```

is modeled as $\lambda v.\lambda self.S(self)$ `||<`$m = e$`>`. It should be noted that S is a function from *self* to a record (object) and since the *self* of C was passed as a parameter to S, any reference to *self* in class S during the definition of C refers to the *self* of C, not S.

In the usual models of object-oriented languages, selective inheritance can be easily obtained. The construct:

```
class (v) inherits i from S (e1, e2) methods j=e3, k=e4 end
```

can be modeled as

$$\lambda v.\lambda self.\texttt{<}i = S(e_1, e_2)(self).i\texttt{>||<}j = e_3, k = e_4\texttt{>}$$

Since $S(e_1, e_2)(self)$ is an object (a record), the specified fields are inherited using record selection.

4 An alternative model

In this section we present a new model of object-oriented languages which is more permissive with regard to static typing than the usual models. Instead of modeling a class as a function from *self* to a record we make each method a function of *self* individually. For the sake of clarity we illustrate the translation with several separate expressions, rather than one convoluted expression that captures all possible combinations of selective and class inheritance. This translation is presented in figure 1.

The generalization of the translation to multiple `inherits` clauses is left to the imagination.

Some explanation is required for `new` and the "methods application" operator. The application $C(e)$ first binds all the instance variables. At this point each method is a function of *self*. By applying all these functions to the same variable, we are requiring that all the individual *self*'s have the same type. The result is one function from *self* to a record. Now the fixpoint operator is applied. This requires that the type of *self* match the type of the record.

Syntax:	$\texttt{class (}v_1\texttt{, ..., }v_n\texttt{) methods }m_1\texttt{=}e_1\texttt{, ..., }m_k\texttt{=}e_k\texttt{ end}$
Translation:	$\lambda v_1. \ldots . \lambda v_n. \texttt{<}m_1 = \lambda self.\, e_1, \ldots, m_k = \lambda self.\, e_k\texttt{>}$
Syntax:	$\texttt{class (}v\texttt{) inherits all from }S\texttt{ (}e\texttt{) methods }m\texttt{=}e'\texttt{ end}$
Translation:	$\lambda v.\, S(e)\texttt{||<}m = \lambda self.\, e'\texttt{>}$
Syntax:	$\texttt{class (}v\texttt{) inherits }m\texttt{ from }S\texttt{ (}e\texttt{) methods }n\texttt{=}e'\texttt{ end}$
Translation:	$\lambda v.\, \texttt{<}m = S(e).m\texttt{>||<}n = \lambda self.\, e'\texttt{>}$
Syntax:	$\texttt{class (}v\texttt{) inherits }m\texttt{ from }S\texttt{ (}e\texttt{)}$ $\quad\texttt{inherits }p\texttt{ from }P$ $\quad\texttt{methods }n\texttt{=}e'\texttt{ end}$
Translation:	$\lambda v.\, \texttt{<}m = S(e).m\texttt{>||<}p = P.p\texttt{>||<}n = \lambda self.\, e'\texttt{>}$
Syntax:	$\texttt{class (}v\texttt{) inherits all from }P\texttt{ (}e_1, e_2\texttt{)}$ $\quad\texttt{inherits }m\texttt{ from }S\texttt{ (}e\texttt{)}$ $\quad\texttt{methods }n\texttt{=}e'\texttt{ end}$
Translation:	$\lambda v.\, \texttt{<}P(e_1, e_2)\texttt{>||<}m = S(e).m\texttt{>||<}n = \lambda self.\, e'\texttt{>}$
Syntax:	$\texttt{new (}C\texttt{, }e\texttt{)}$
Translation:	$\mu(\lambda v.C(e)@v)$

Figure 1: Translation of object-oriented constructs

The new model provides greater flexibility in the typing of inheritance where methods have been overridden or selectively inherited and in the typing of abstract classes as we will show in the following sections.

The following example is intended to present the notation used in this paper in a simple setting. It also serves to illustrate the line of reasoning that occurs in most of the subsequent examples. It is not intended to be indicative of the differences between the model presented in this paper and the usual models. Consider the class

```
class (x) methods m=4+x+self.l end
```

The class, call it C, is modeled (assuming the model presented in this paper is being used) by the following expression:

$$\lambda\, x\, .\, \texttt{<}m = \lambda self.\, 4 + x + self.l\texttt{>}$$

Even without being specific about the type system for expressions, we can say that C has roughly the type:

$$integer \rightarrow \texttt{<}m : S \rightarrow integer\texttt{>}$$

where S (the type of *self*) is the type $\texttt{<}l : integer\texttt{>}$. While types for S that have additional fields can also be derived, all derivable types of S must have an l-field of type *integer*. The expression $N = \lambda\, v\,.\, C(3)@v$ has the following type:

$$\texttt{<}l : \mathit{integer}\texttt{>} \rightarrow \texttt{<}m : \mathit{integer}\texttt{>}$$

Any type that can be derived for N has a range that is a record with only one field: the m-field; the range can have no other fields. Types can be derived for N where additional fields in the domain are present, but the domain must always have an l-field of type *integer*.

Note that for the fixpoint to be applied the domain and range must have the same type, but in this case that cannot happen; the domain must have an l-field and the range cannot have an l-field. Since instantiation requires an application of the fixpoint operator the class C cannot be instantiated. It should be pointed out that the fixpoint cannot be applied to any supertype of any derivable type of N either.

4.1 Selective inheritance

The key feature with respect to the typing of selective inheritance in the new model is that the *self* of the subclass is not an argument to the superclass. This eliminates the possibility that methods not inherited by a subclass affect the type of that subclass, except with regard to the instance variables. The one exception to the previous statement is with regard to instance variables. This will be discussed in greater detail in the conclusion. To see how the passing of the *self* of the subclass as an argument to the superclass affects the type of the subclass, consider the situation where a class C inherits a method from the class S. In the usual models $S(\mathit{self})$ is a subexpression of the translation of class C. Thus, $S(\mathit{self})$ must be typable for class C to be typable. This means that the *self* of C must meet all type demands imposed by its superclass S. If method m in S demands that method k have type integer, then the type of the *self* of C must have a k field of type integer, regardless of whether C has inherited methods m or k from S.

Under the model we have proposed, a method m would be inherited from class S with the code $S.m$ rather than $S(\mathit{self}).m$. The only type demands that the expression $S.m$ imposes is that S be a record having an m field. In contrast, the expression $S(\mathit{self}).m$ demands that S have the form $\lambda v.e$, that e be a record having an m field, and furthermore that *self* satisfy all the type restrictions that e places on the formal parameter v.

To illustrate this situation further, consider the superclasses P and Q given by

```
class () methods area=e₁, vol=e₂, cost=e₃, mass=e₄ end
class () methods fuel=e₅, emi=e₆, cost=e₇, value=e₈ end
```

respectively, and the subclass C given by

```
class ()
    inherits area, vol, cost from P
    inherits fuel, emi from Q
end
```

The translation of classes P, Q and C under the usual models are as follows:

$$P = \lambda self.\,\texttt{<}area = e_1,\ vol = e_2,\ cost = e_3,\ mass = e_4\texttt{>}$$
$$Q = \lambda self.\,\texttt{<}fuel = e_5,\ emi = e_6,\ cost = e_7,\ value = e_8\texttt{>}$$
$$C = \lambda self.\,\texttt{<}area = P(self).area,\ vol = P(self).vol,$$
$$cost = P(self).cost,\ fuel = Q(self).fuel,$$
$$emi = Q(self).emi\texttt{>}$$

Assume that the methods $fuel$, emi in class Q are independent of the methods $cost$ and $value$, but that the method $value$ contains the subexpression $self.cost$ in such a way that the $cost$ method must have type $real$. Let us further assume that in class P the method $cost$ has type $integer \rightarrow integer$.

Using the usual models, the class C cannot be instantiated. The subexpression $Q(self)$ present in the modeling of C demands that Q be consistent with the $self$ of C, and so the $cost$ method of $self$ must have type $real$. However, the $cost$ method of C is taken directly from P and so has type $integer \rightarrow integer$. This means that C is a function type from records to records in which the domain contains a $cost$ field having type $real$ and the range contains a $cost$ field having type $integer \rightarrow integer$. But, the typed fixpoint operator is not applicable to this type (nor to any supertype), so C cannot be instantiated.

Under the usual modeling of inheritance, the $value$ field of Q renders C uninstantiable, in spite of the fact that this method is not even present in C. This is an unnecessary type restriction. The $cost$ and $value$ fields of Q have nothing to do with C and should not render it uninstantiable. If P, Q, and C are modeled in the manner we have proposed

$$P = \texttt{<}area = \lambda self.e_1,\ vol = \lambda self.e_2,\ cost = \lambda self.e_3,$$
$$mass = \lambda self.e_4\texttt{>}$$
$$Q = \texttt{<}fuel = \lambda self.e_5,\ emi = \lambda self.e_6,$$
$$cost = \lambda self.e_7,\ value = \lambda self.e_8\texttt{>}$$
$$C = \texttt{<}area = P.area,\ vol = P.vol,\ cost = P.cost,$$
$$fuel = Q.fuel,\ emi = Q.emi\texttt{>}$$

respectively, then class C is instantiable. There is no $Q(self)$ expression in the translation of C using our model, hence the demand that Q be typable with the $self$ of C has been eliminated. Therefore, the uninherited methods of Q, $cost$ and $value$, do not affect the type of C.

4.2 Overriding methods

The ability to define classes that can be instantiated only under the new model is not restricted to classes that use selective inheritance. The same principles apply when inherited methods are overridden (redefined) by the subclass. This can be illustrated with an example similar to the one used in the discussion on selective inheritance. Let S and C be the following classes

$S =$ `class ()`
`methods`
    `fuel=`e_1`, emi=`e_2`, cost=`e_3`, value=`e_4
`end`

$C =$ `class () inherits all from` S
`methods`
    `cost=`e_5`, value=`e_6
`end`

where again $fuel$ and emi are not dependent on the methods $cost$ and $value$, but the code for $value$ of class S requires that the $cost$ method of class S have type $real$. We will also assume that the method $cost$ defined in class C does not have type $real$. For the same reasons as before, C is typable under the usual models, but cannot be instantiated. C is modeled as $\lambda self.\texttt{<}S(self)\texttt{>||<}cost = e_5, value = e_6\texttt{>}$. The $S(self)$ subexpression demands that the domain of C be a record having a $cost$ field of type $real$ (due to the requirements imposed by the $value$ field of S) while the range of C is a record whose $cost$ field does not have type $real$. The $value$ field of S which was not present in C, since it was overridden, still prevented C from being instantiable.

However, the class C is instantiable under the new model. While the $value$ field of S is inherited by C, it does not affect the type of C under this model. Since each method has its own version of $self$, the methods of a class do not affect the types of the other methods of that class (except with regard to instance variables). Thus, the $value$ field of S does not affect the types of the other methods during the typing of S. The k field of S is then overridden by the k field defined in C at which point no type information from the k field of S remains in the type of C.

This example is a distillation of an example used to show the usefulness of subtype bounded polymorphism [7]. In that paper they consider a class S

`class () methods i=5, eq =` λ `o. (o.i=self.i) end`

and subclass C

```
class () inherits P() methods b=true, eq = λ o.
(o.i=self.i)&(o.b=self.b) end
```

4.3 Abstract classes

The model proposed in this paper also imposes fewer type restrictions on abstract classes than the previous models. We do not claim this as an advantage or disadvantage of the proposed model. We simply make note of the difference with respect to the previous models.

An abstract class is a class that defers some method definitions to its subclasses. This is useful in abstracting out the methods that several classes may have in common [11]. Instances of abstract classes may never be, and in some cases cannot be, instantiated. For a class to be instantiable it must be possible to derive a type with the same domain and range. The instantiability of a class is an issue only when an object is instantiated by the `new` operation, not when the class is defined. Both the usual models and the model proposed in this paper permit uninstantiable classes to be used as abstract classes.

The following class, C,

```
class () methods i=self.k or true, j=self.k+1 end
```

is an example of an abstract class. Since any type for C must have a domain that is a record type with a `k` field (due to the term `self.k` in method `i`) and the range cannot have a `k` field as there is not one defined, C does not have a fixpoint. Thus, C cannot be instantiated. It is the responsibility of the subclasses of C to define a `k` field. While C cannot be instantiated using either the usual models or the model proposed in this paper, the new model permits C to be used as an abstract class.

If C is translated to

$$\lambda self.\texttt{<}i = self.k \; or \; true, \; j = self.k + 1\texttt{>}$$

as per the usual models, then it cannot be typed. There is no type which can be given to *self* that permits both the `i` field and `j` field to be typed due to their conflicting demands on the type of the `k` field of *self*. However, if C is translated to

$$\texttt{<}i = \lambda self \, .self.k \; or \; true, \; j = \lambda self \, .self.k + 1\texttt{>}$$

as in the new model, then it can be typed (but not instantiated). The reason it can be typed is that the `i` field and `j` field each have their own version of *self*. The `i` field demands that *self* have a boolean `k` field.

If a class has no superclasses, then it is either instantiable under both models or neither model. The new model is more permissive in typing classes, but a class that can only be typed using the new model can only be used as an abstract class—it cannot be instantiated. However, we see no major advantage in being able to have abstract classes consisting of incompatible methods nor do we consider the possibility of these classes a major drawback.

4.4 Example

Here, we present one last example illustrating the added flexibility of the model we have proposed. This example does not add anything conceptually to the examples previously presented in this section—it serves as motivation of selective inheritance, not as an explanation of it. It is a more practical example of the usefulness of the additional flexibility of our model and of selective inheritance.

First, consider the union-find problem on a restricted universal set. A tree structure, implemented with an array, provides an efficient data structure for the problem. Assume the class *UnionFindRestricted* that implements this data structure has already been written. Among its methods are the following (and their associated types):

```
Union:  SetName*SetName*table -> table
Find:   element*table -> SetName
```

where `table` is simply a record of the arrays used in the data structure and `element` is an integer in the range 0 to n.

We will also assume that there is a class *HashTable* that implements a hash table with open hashing and hashes integers to the range 0 to m (where m is specified during initialization). Among the methods of *HashTable* are:

```
Insert: element*data*HashTable -> HashTable
Member: element*HashTable -> data
Delete: element*HashTable -> HashTable
Find:   element*HashTable -> pointer
```

where `element` is an integer and `data` is the data associated with each integer. The method `Find` simply returns a pointer to the entry for its given argument. We will also assume that the method `Delete` calls the method `Find` in its code (important in this example).

Assuming that we later need to implement the operations `Union` and `Find` on an unrestricted universal set (say the universal set is the set of integers), then the data structure used in class *UnionFindRestricted* is not sufficient. The implementation of `Find` relies on the ability to use the element as an index into the array, and that ability is lost when the universal set is not restricted. However, we could combine the data structures used in class *UnionFindRestricted* with a hash table to implement union-find on the set of integers. All that is needed is for every element (integer) in each set to be placed in a hash table along with an integer in the range 0 to n (which serves as an index into the array in *UnionFindRestricted*). To execute the `Find` operation on some integer, we look it up in the hash table (to determine its index in the data structure implemented in *UnionFindRestricted*) and then `Find` from *UnionFindRestricted* is called with this index as an argument.

Therefore, we can define a class *UnionFind* that inherits the methods `Union`, `Find` from *UnionFindRestricted* and `Member` from *HashTable*. However, the class *UnionFind* cannot be instantiated under the usual models of inheritance. In these models there would be the subexpression *HashTable*(*self*) in the definition. The *self* of *UnionFind* must satisfy all type constraints placed upon it by the methods of *HashTable*. The method `Delete` which was not inherited by *UnionFind* calls the method `Find` (the one in *HashTable*) and so the *self* of *UnionFind* must have a method `Find` with the type `element*HashTable -> pointer`. However, *UnionFind* also has a `Find` method of type `element*table -> SetName` since `Find` was inherited from *UnionFindRestricted*. Thus, there is no fixpoint due to the conflicting types of the `Find` method—even though the `Find` method from *HashTable* was not inherited nor was any method that called the `Find` method of *HashTable*.

In the model we propose in this paper, the class *UnionFind* can be instantiated since the types of uninherited and overridden methods of a superclass do not affect the type of the subclass.

The ability to define classes and not have their type affected by uninherited and overridden methods would be most useful in larger software systems in which a large number of classes have been defined. For example, consider the situation where there is a long chain of inheritance relationships with methods being overridden again and again. Under the usual models, the type demands imposed by these methods are still present in the types of the subclasses and the programmer must avoid defining methods that conflict with the demands of these methods (that are not present in the class that is being defined). Furthermore, these methods may not even be present in the superclasses that are explicitly specified in the definition.

5 Subclasses and subtypes

In this section we have a short discussion about the approach of obtaining typing rules for object-oriented languages using models. We need not consider a *particular* type system, because the remarks apply to the various type systems proposed for object-oriented languages.

While an object-oriented language can be defined by models, the type system for the language need not be defined by viewing the various constructs as macros and using the typing rules of the typed lambda calculus. Typing rules can be defined for the object-oriented constructs, such as those in section 2, without regard to how these constructs are translated into the lambda calculus. If the typing rules are obtained in this fashion, then it is convenient to demand a subclass be a subtype of its superclasses. One reason for making such a restriction can be seen by the existence of the $S(\mathit{self})$ subexpression in the model of inheritance, where the actual parameter *self* refers to the *self* of the subclass. If subclasses must be subtypes of their superclasses, then the expression $S(\mathit{self})$ will always be type correct. To see this, assume that C is a subclass of S. If class C is also a subtype of S, then it is ensured that the expression $S(\mathit{self})$, where *self* refers to C, is type correct, since the basic idea of a subtype is that it can be used anywhere one of its supertypes is used.

While an enforced subtype relation between class and superclass is sufficient to ensure type safety, it is not necessary. To see this, consider the classes S and C given by

`class ()` m`=4,` n`=5 end`

`class () inherits all from` S `methods` m`=true end`

which are modeled as: $\mathrm{S}{=}\lambda \mathit{self}.\texttt{<}m = 4, n = 5\texttt{>}$ and $\mathrm{C}{=}\lambda \mathit{self}.S(\mathit{self})\texttt{||<}m = \mathit{true}\texttt{>}$, respectively. There are no demands placed on *self* in class S. What *self* is bound to is irrelevant in the typing of class S. This means that $S(\mathit{self})$ is type correct regardless of whether C is a subtype of S.

While a subtype relation between subclass and class is not necessary for type safety, such a relationship has the advantage that it is easy to check. If the subtype relationship exists, then the $S(\mathit{self})$ expression is type correct. The drawback to demanding a subtype relation between class and subclass is that it is unnecessarily restrictive, as seen in the previous example as well as in [7]. As another example of the unnecessary restrictiveness of the demand for a subtype relation between class and subclass consider the classes P and Q

`class () methods` $m = e_1$`,` $n = e_2$`,` $o = e_3$`,` $p = e_4$ `end`

`class () methods` $a = e_5$`,` $b = e_6$`,` $c = e_7$`,` $p = e_8$ `end`

respectively. If e_4 and e_8 do not have a mutual subtype, then no class can inherit methods from both P and Q since they have no mutual subtype. The demand that a subtype relation exist between class and subclass prohibits multiple inheritance from two classes that have a method name conflict (where the methods in question have different types) as well as prohibiting the overriding of methods with an expression having a different type.

If the object-oriented constructs are viewed as macros for typed lambda expressions and typing rules for the constructs are obtained using the typing rules of the typed lambda calculus, then type systems and type checking (or type reconstruction) algorithms for object-oriented languages can be directly obtained from the model, as seen in [17, 19]. Type systems obtained in this fashion place no type restrictions other than those imposed by the typed lambda calculus and the model itself.

6 Conclusion

In this paper we discuss the concept of the selective inheritance of methods and its advantages when multiple inheritance is permitted. It is shown that while the usual models of object-oriented languages are fine for class inheritance, they do not behave in the expected fashion when selective inheritance is permitted. For this reason we propose a new model of object-oriented languages. If typing rules for the object-oriented constructs are obtained from this model, then unnecessary type restrictions that occur under the previous models are avoided. The proposed model is also strictly more lenient with regard to typing rules than the previous models even if restricted to class inheritance.

The main difference in the typing rules obtained from the proposed model and the previous models lies in the interaction of inheritance and typing. In the previous models, the uninherited and overridden methods of a class still affect the types of its subclasses. This means that it is possible for a method that is not present in a class to prevent that class from being used as a template for creating objects. In the model proposed in this paper, the only effect that uninherited and overridden methods can have on the types of subclasses is with regard to instance variables.

References

[1] C. A. G. adn John C. Mitchell, editor. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* Foundations of computing. MIT Press, Cambridge, Massachusetts, 1994.

[2] V. Breazu-Tannen, O. P. Buneman, and C. A. Gunter. Typed functional programming for rapid development of reliable software. In J. E. Gaffney, editor, *Productivity:*

Progress, Prospects and Payoff, pages 115–125, June 1988.

[3] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[4] L. Cardelli. Typeful programming. Technical Report SRC Report 45, Digital Equipment Corporation, Systems Research Center, May 1989.

[5] L. Cardelli. Extensible records in a pure calculus of subtyping. Technical Report SRC Report 81, Digital Equipment Corporation, Systems Research Center, January 1992.

[6] L. Cardelli and J. C. Mitchell. Operations on records. Technical Report SRC Report 48, Digital Equipment Corporation, Systems Research Center, August 1989.

[7] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135. ACM, 1990.

[8] W. R. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In N. Meyrowitz, editor, *Object-Oriented Programming: Systems, Languages and Applications*, pages 433–443. ACM, 1989.

[9] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, Reading, MA, 1990.

[10] L. A. Jategaonkar and J. C. Mitchell. ML with extended pattern matching and subtypes (preliminary version). In *Conference record of the 1988 ACM symposium on LISP and functional programming*. ACM, 1988.

[11] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2), 1988.

[12] B. Meyer. Resuability: The case for object-oriented design. *IEEE Software*, 4(4):50–64, March 1987.

[13] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

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

[15] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 77–88. ACM, 1989.

[16] R. Stansifer. Type inference with subtypes. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 88–97. ACM, January 1988.

[17] M. Wand. Complete type inference for simple objects. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 37–44. 1987.

[18] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 92–97. 1989.

[19] M. Wand. Type inference for objects with instance variables and inheritance. Technical Report NU-CCS-89–2, College of Computer Science, Northeastern University, November 88.

[20] P. Wegner. The object-oriented classification paradigm. In P. Wegner and B. D. Shriver, editors, *Research directions in object-oriented programming*, pages 479–560. MIT Press, Cambridge, MA, 1987.

[21] D. Wetklow. *Type reconstruction algorithms for object-oriented languages*. PhD thesis, Purdue University, 1990.