1991

# Selective Inheritance and Overriding in Statically-Typed Object-Oriented Languages

Ryan Stansifer

Dan Wetklow

Report Number:
91-053

# SELECTIVE INHERITANCE AND OVERRIDING IN STATICALLY-TYPED OBJECT-ORIENTED LANGUAGES

Ryan Stansifer
Dan Wetklow

CSD-TR-91-053
July 1991

# Selective Inheritance and Overriding in Statically-Typed Object-Oriented Languages

Ryan Stansifer
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Dan Wetklow
Department of Computer Science
University of Pittsburgh at Johnstown
Johnstown, PA 15956

July 24, 1991

**Abstract**

We examine inheritance in statically-typed object-oriented languages. In some object-oriented languages it is possible to inherit selected methods from superclasses. We call this 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. Furthermore, our approach allows more classes to be typed in cases where methods are overridden. We discuss modeling object-oriented languages and the implications of our proposed model.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs—*data types and structures*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*
General Terms: Languages.
Additional Keywords and Phrases: Object-oriented languages, inheritance, records

## 1 Introduction

This paper is about inheritance in object-oriented languages. It is no small matter to explain what this means. We make no attempt to say what is, or what is not, an "object-oriented" language. For an interesting discussion of this issue see the paper by Wegner [17]. Instead we adopt a formal framework used by others which captures aspects of object-oriented languages and inheritance. Differences are possible in how inheritance is understood in a formal framework. Obviously, the formalism may, or may not capture what someone thinks "inheritance" should mean. It may, or may not, have the computational semantics that "inheritance" should have. It may, or may not, have the right combination of compile-time properties. This last topic is the concern of this paper.

To examine the compile-time properties of inheritance we must first have a rigorous definition of the language. One manner of obtaining such a definition is by translating the desired object-oriented constructs to the well-studied language of lambda expressions. We refer to such a translation as a *model*. In section 2 we suggest some desirable object-oriented constructs, and give a provisional syntax. In section 3 we look at existing models for object-oriented languages. In section 4 we propose a slightly different model and discuss its advantages.

The record-based model of object-oriented languages introduced by Cardelli [2] provides a framework for strongly typed object-oriented languages. This model has played a role in languages like Modula-3 [4] and is not unlike the approach taken in C++ [8]. 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 value of types in programming languages can be found in several places [1, 3]. A major advantage of a strongly typed object-oriented language is the elimination of the "message not understood" error.

In this paper we present a new model of object-oriented languages that imposes fewer type restrictions on subclasses than previous models [6, 11]. The natural typing rules can be used to type more expressions in the new model than in the previous models. The main feature of the new model is that the methods of a superclass that are not inherited by the subclass or are overridden by the subclass do not effect the type of the subclass. This result is relatively independent of the type system used.

# 2 Modeling Object-Oriented Languages

Among the distinguishing features of object-oriented languages is that data is 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 some hypothetical language is given, and these constructs are explained informally. In section 3 we make the semantics of these constructs precise by giving a translation of them to typed lambda-expressions.

## 2.1 Object-Oriented Languages

For a class we introduce the `class` construct and use the following syntax:

`class` $(v_1, \ldots, v_n)$ `methods` $m_1$=$e_1$, ..., $m_k$=$e_k$ `end`

A class with this definition has instance variables $v_1, \ldots, v_n$, where $n$ may be 0. These 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` $(C, e_1, \ldots, e_n)$

will denote this mechanism, where $C$ denotes some class and $e_1, \ldots, e_n$ are expressions denoting 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. Different instances of a class may have different values for the instance variables by invoking the `new` operation on the class with different instance values.

Methods of an object $O$ are accessed by $O.m$ where $m$ denotes one of $O$'s methods. Static typing ensures that correctly typed expressions of the language access methods that are in the protocol of the objects. 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.

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. This is what Wegner [17, page 509] calls class inheritance. For example, let $C$ be the class

`class` $(v)$ `inherits all from` $S$ $(e_1, \ldots, e_2)$ `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 $S$.

The `inherits all` construct passes all the methods to the subclass. We add in our object-oriented language a related construct to allow the subclass to inherit just 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 to inherit all of the methods of the superclass, and one to inherit only the methods of the superclass that are specified.

More than one `inherits` clause is permitted. A class can have, therefore, multiple superclasses. This is called multiple inheritance. In the case of multiple inheritance especially, it seems reasonable to allow the selective inheritance of methods. This way if more than one superclass provides a method, the subclass can explicitly name which superclass is to provide the method. Although not an issue here, multiple inheritance does cause problems with type reconstruction [14, 16, 18].

We summarize, in the table below, some of the relevant object-oriented terminology.

| | |
|---|---|
| `class` | construct to encapsulate set of methods |
| object | instance of class |
| `new` | construct for creating an object from a class |
| `self` | pseudo variable referring to the object itself |
| message passing | request that method be invoked |
| inheritance | obtaining methods from another class |
| selective inheritance | inheriting named methods |
| superclass | a class from which a class inherits methods |
| multiple inheritance | more than one superclass |

## 2.2 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 |
| $\mu e$ | fixpoint |
| $<m_1 = e_2, \ldots, m_k = e_k>$ | records |
| $e.m$ | field selection |
| $e_1 \mid\mid e_2$ | record concatenation |
| $e_1 \mathbin{@} e_2$ | "apply to" |

In addition to the usual constructs of the lambda calculus we have records, finite associations of values to labels. We write records using angle brackets, e.g., $<a = 4, b = true>$. The expression, $e_1 \mid\mid e_2$, is record concatenation, by which we mean the fields of record $e_2$ are added to those of record $e_1$. For example, $<a = 4, b = v> \mid\mid <c = 9>$ is the record $<a = 4, b = v, c = 9>$. In the case of a name conflict among labels, the field of the second record takes precedence. So, $<a = 4, b = v> \mid\mid <a = true>$ is the record $<a = true, b = v>$. Record concatenation is a different construct from record extension [5] which requires that no field of the second record appear in the first.

The last record construct in the table is not used in other models of object-oriented languages. It is added explicitly for the model presented in this paper. We refer to it as the "apply to" operation. It takes two arguments, an arbitrary expression and a record that has the property that every field in the record is a function. The result of this operation is the record obtained by applying every field to the expression. For example, the expression $<a = \lambda v.v, b = \lambda v.\lambda x.x, c = \lambda y.<m = y>>$ @ 3 has the value $<a = 3, b = \lambda x.x, c = <m = 3>>$. The "apply to" operator is only used in modeling the `new` construct.

### 2.3 Types

While the focus of this paper is on the interaction of the demands of static typing and the modeling of inheritance, we do not give a particular type system for the typed lambda expressions given in the previous section. Type systems and type reconstruction algorithms for the typed lambda calculus augmented with record constructs have been studied [9, 12, 13, 14, 18]. These relatively complex type systems detract from the purpose here of discussing models of inheritance. All we need assume here are:

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

2. the usual type rules for function application, the fixpoint operator, and so on.

A rule for record concatenation is somewhat problematic, but we need not get specific.

In typed models of object-oriented languages the subtype relation is important. The type $\tau_1$ is a subtype of the type $\tau_2$, denoted $\tau_1 \leq \tau_2$, if an expression having type $\tau_1$ can be used anywhere that an expression having type $\tau_2$ can be used, i.e., there is no danger of a run-time type error. For this reason it can be said that an expression having type $\tau_1$ also has type $\tau_2$. A reasonable type system permits expressions of type $\sigma$ to have type $\tau$ if $\sigma \leq \tau$. 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.

## 3 The usual model

In a seminal work by Cardelli [2] 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.

### 3.1 Records

The class construct:

`class` $(v_1, \ldots, v_n)$ `methods` $m_1$=$e_1$, ..., $m_k$=$e_k$ `end`

can be modeled as follows:

$$\lambda v_1. \ldots . \lambda v_n . <m_1 = e_1, \ldots, m_k = e_k>$$

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`

More refined models [6, 11] have been put forth which capture the purpose of the pseudo variable *self*. The methods of an object can refer to other methods, or to the object itself, via *self*. For example, the methods of some hypothetical class $m_1 = 3$, and $m_2 = self.m_1 + 1$ 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` $(v_1, \ldots, v_n)$ `methods` $m_1$=$e_1$, ..., $m_k$=$e_k$ `end`

is modeled as

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

The creation of an object from a class is not a simple function application which binds the values of all the instance variables. Creating an object requires taking the fixpoint of a class. For 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 made to refer to the object as a whole. For the sake of simplicity, we generally will not bother with instance variables in class and object declarations.

In addition to *self*, inheritance is also accounted for in the standard model of object-oriented languages and is modeled as: $C = \lambda self.S(self)||\langle m = e\rangle$ if $S$ is a class with no instance variables. It should be noted that $S$ is a function from *self* to a record (object) and that 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 $S$.

Selective inheritance can be added. The construct:

`class (v) inherits i from S (`$e_1$`, `$e_2$`) methods j=`$e_3$`, k=`$e_4$` end`

can be modeled as $\lambda v.\lambda self.\langle i = S(e_1,e_2)(self).i\rangle||\langle j = e_3, k = e_4\rangle$. Since $S(e_1,e_2)(self)$ is an object (a record), the specified fields can be inherited using record selection.

## 3.3 Subclasses and subtypes

While an object-oriented language can be defined by the standard model, 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. In such cases it is convenient to demand a subclass be a subtype of its superclasses. One reason for making such a restriction is the existence of the $S(self)$ 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 $S(self)$ 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 $S(self)$, 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: S=$\lambda self.\langle m = 4, n = 5\rangle$ and C=$\lambda self.S(self)||\langle m = true\rangle$, 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 $S(self)$ 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 $S(self)$ 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].

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 [14, 15]. Type systems obtained in this fashion place no type restrictions other than those imposed by the typed lambda calculus and the model itself.

## 3.4 Abstract classes

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 [10]. Instances of abstract classes may never be, and in some cases, cannot be instantiated.

The demand that a class be instantiable is more stringent than the demand that a class be type-correct. For a class to be type-correct, it must be possible to derive a type for it (any type will suffice). Note that the type of a class will be a function type. 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.

# 4 An alternative model

In this section we present a new model of object-oriented languages. This model has certain advantages over the usual model when static typing is desired.

The essential idea is that instead of modeling a class as a function from *self* to a record, we make each method a function of *self* individually. A few examples will make the translation clear and we do not bother with a more formal exposition of the translation.

Syntax: `class` $(v_1, \ldots, v_n)$ `methods` $m_1$=$e_1$, ..., $m_k$=$e_k$ `end`
Translation: $\lambda v_1. \ldots . \lambda v_n . <m_1 = \lambda self . e_1, \ldots, m_k = \lambda self . e_k>$

Syntax: `class` $(v)$ `inherits all from` $S$ $(e)$ `methods` $m$=$e'$ `end`
Translation: $\lambda v . S(e) || <m = \lambda self . e'>$

Syntax: `class` $(v)$ `inherits` $m$ `from` $S$ $(e)$ `methods` $n$=$e'$ `end`
Translation: $\lambda v . <m = S(e).m> || <n = \lambda self . e'>$

Syntax: `new` $(C, e)$
Translation: $\mu(\lambda v . C(e)@v)$

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

Some explanation is required for `new` and the "apply to" operator. The application $C(e)$ first binds all the instance variables. At this point each method is a function from *self* to whatever. 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.

The new model provides greater flexibility in the typing of abstract classes and inheritance where methods have been overridden or selectively inherited. This flexibility will be illustrated in the following sections. In the context we take up first, uninstantiable abstract classes, we see the sharpest contrast. The other contexts are more useful in practice, these are examined afterward.

## 4.1 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 model or the model proposed in this paper, the new model permits $C$ to be used as an abstract class.

If $C$ is translated to

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

as per the standard model, 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

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

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.

It is easy to prove that if a class has no superclasses, then it is either instantiable under both models or neither model. The new model is strictly more lenient with regard to the typing of 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.

## 4.2 Selective inheritance

The key feature with regards 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. 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 model, $S(self)$ is a subexpression of the translation of class $C$. Thus, $S(self)$ 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 $S$.

To illustrate this situation further, consider the superclass $S$ given by

```
class () methods i=3, j=self.i, k=self.i+1 end
```

and two subclasses $C$ and $D$ given by

```
class () inherits j from S methods i=true end
class () inherits j,k from S methods i=true end
```

respectively. The translation of class $S$ under the standard model is

$$\lambda self.\langle i = 3, j = self.i, k = self.i + 1\rangle$$

and is typable and instantiable in any sort of reasonable type system. The pseudo variable `self` is a record with an `i` method of type `integer`. When the class $C$ is modeled as

$$\lambda self.\langle j = S(self).j\rangle || \langle i = true\rangle$$

its type is problematic. It is reasonable to derive the type

$$\langle i : int\rangle \rightarrow \langle i : bool, j : bool\rangle$$

for $C$, i.e., the function type from records to records in which the domain contains an $i$ field of integer type and the range contains an $i$ field of boolean type. But the typed fixpoint operator is not applicable to this type (or to any supertype). So the class cannot be instantiated.

The class $D$ suffers the same type problems as $C$ in that it is typable, but not instantiable. But $D$, by inheriting both of the methods `j`, `k`, should not be instantiable since the `i` field and `k` field place conflicting demands on the type of `self`. On the other hand, it is reasonable that the class $C$ should escape this problem. Class $C$ can be viewed as computationally equal to

```
class () methods j=self.i, i=true end
```

and this class ought to be instantiable. If we model class $S$ in the manner we have proposed

$$\langle i = \lambda self.3, j = \lambda self.self.i, k = \lambda self.self.i + 1\rangle$$

and class $C$ as

$$\langle j = S.j\rangle || \langle i = \lambda self.true\rangle$$

then $C$ is instantiable. The key is that the uninherited method `k` does not affect the type of class $C$ and therefore the conflicting type demands placed on the `i` field under the standard model do not occur.

The main difference between the two models is that the new model delays the point at which the types of `self` for the individual methods are made to agree until the class is instantiated. Instantiation (the `new` operation) under the new model is conceptually the same as in the standard model; essentially demanding that a fixpoint exist. The `new` operation under the model proposed in this paper first demands the types of *self* for the individual methods of the class agree, then demands there exist a fixpoint.

### 4.3 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. Consider the following classes:

$S$ =`class () methods i=3, j=self.i, k=self.i+1 end`
$C$ =`class () inherits all from` $S$ `methods i=true, k=true end`

$C$ is typable under the standard model, but cannot be instantiated. The `k` field of $S$ demands that any type of $C$ have a domain where the `i` field has type integer, and from the definition of $C$ it is seen that in the range of any type of $C$ the `i` field must have type boolean. Thus, there is no fixpoint.

However, the class $C$ is instantiable under the new model. While the `k` field of $S$ is inherited by $C$, it does not affect the type of $C$ under the model proposed in this paper. 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 regards to instance variables). Thus, the `k` 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 $C$.

## 5 Conclusion

We are willing to let classes have types for the purposes of inheritance even when uninstantiable.

We have presented a new model of object-oriented languages. While this model is computationally the same as the standard model, some differences emerge are far as type-checking is concerned. By viewing the models of object-oriented constructs as macro definitions, typing rules can be obtained for these constructs from the typing rules for lambda expressions (augmented with record constructs). The typing rules that are obtained using the model presented in this paper are more lenient than those obtained from the standard model. This is independent of the type system being used.

The main difference in the typing rules obtained from the two models lies in the interaction of inheritance and typing. In the standard model, 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 presented in this paper, the only effect that uninherited and overridden methods can have on the types of subclasses is with regard to instance variables. We felt that the types of the instance variables should be consistent in all of the methods of the class. Giving each method its own copy of the instance variables (as was done for *self*) and then using the "apply to" operator to demand agreement during instantiation is all that is required to modify the model to eliminate the requirement that the types of the instance variables should be consistent in all the methods of a class. This increases the complexity of the type of a class without much benefit in return.

Using type rules for inheritance based on the types of *objects* is very restrictive. A Allowing classes to have types seems more flexible and obtaining subclasses which are are not subtypes is easy.

Although we have focused on types it is worth while to point out that in any of the models of *self* implementing objected instantiation is difficult. It requires applying the fixpoint to an arbitrary expression. Typically languages restrict the fixpoint operation to function definitions.

**Acknowledgments.** The authors wish to express their appreciation to Mike Beaven for reading earlier drafts of this paper and helpful discussions.

## References

[1] Breazu-Tannen, Val, O. Peter Buneman and Carl A. Gunter. Typed functional programming for rapid development of reliable software. In *Productivity: Progress, Prospects and Payoff*, J. E. Gaffney (Ed.), (June 1988), 115–125.

[2] Cardelli, Luca. A semantics of multiple inheritance. *Information and Computation*, 76, (1988), 138–164.

[3] Cardelli, Luca. Typeful programming. SRC Report 45, Digital Equipment Corporation, Systems Research Center, May 1989.

[4] Cardelli, Luca, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow and Greg Nelson. Modula-3 report (revised). SRC Report 52, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, November 1989.

[5] Cardelli, Luca and John C. Mitchell. Operations on records. SRC Report 48, Digital Equipment Corporation, Systems Research Center, August 1989.

[6] Cook, William R. and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Object-Oriented Programming: Systems, Languages and Applications*, Norman Meyrowitz (Ed.), ACM, 1989, 433–443.

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

[8] Ellis, Margaret A. and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, Reading, MA, 1990.

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

[10] Johnson, Ralph E. and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 2, 1, (1988).

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

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

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

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

[15] Wand, Mitchell. Type inference for objects with instance variables and inheritance. NU-CCS-89-2, College of Computer Science, Northeastern University, November 1988.

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

[17] Wegner, Peter. The object-oriented classification paradigm. In *Research Directions in Object-Oriented Programming*, Peter Wegner and Bruce D. Shriver (Ed.), MIT Press, Cambridge, MA, 1987, 479–560.

[18] Wetklow, Dan. Type reconstruction algorithms for object-oriented languages. Ph.D. dissertation, Purdue University, Department of Computer Sciences, 1990.