1990

# A statically-typed language with classes

Mike Beaven

Ryan Stanisfer

Dan Wetklow

Report Number:
90-996

# A STATICALLY-TYPED LANGUAGE WITH CLASSES

Mike Beaven
Ryan Stansifer
Dan Wetklow

CSD-TR-996
July 1990

# A statically-typed language with classes

Mike Beaven Ryan Stansifer Dan Wetklow
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

July 18, 1990

# Contents



# List of Figures

## Abstract

We describe the experience of designing a statically-typed, functional language with object-oriented features based on the record model of objects. Some of the distinguishing features of the language include implicit instantiation of objects and selective method inheritance. A prototype implementation of the language has been built. We discuss the effect the type system has on the language, and how it compares with subtype-bounded polymorphism. We describe how the type reconstruction algorithm works on inheritance.

# 1 Introduction

We are interested in statically-typed languages for all the usual reasons: including 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 [2, 4]. If the programmer explicitly declares the type of each identifier, it is usually not difficult to do the type checking. However, most of the time the compiler could determine the types of identifiers from the way they are used in the program. Thus the compiler could spare the programmer the burden of explicitly giving the types. The programming language ML [11] owes some of its appeal to the fact that it can determine the types of all the identifiers.

The record-based model of object-oriented languages introduced by Cardelli [3] provides a framework for a strongly typed object-oriented language. An advantage of a strongly typed object-oriented language is the elimination of the "message not found" error. We are interested in the problem of type reconstruction for such languages. Numerous efforts have been attempted to devise a satisfactory type reconstruction algorithm for subtype polymorphism [8, 9, 13, 12, 17, 18]. Many of these studies have type systems that are complex. One promising direction is using parametric polymorphism to obtain subtype polymorphism. This is the approach we take in the language described in this paper.

We have built a prototype implementation of the language in ML. A more detailed description of a preliminary version of the language and the implementation can be found in [1]. This implementation allows us to gain experience with how the type system affects the design of an object-oriented language. One goal has been to make the types of objects as simple as possible. We are also motivated to maintain the ability to do type reconstruction with selective inheritance. In this paper we see how these goals clash.

The next section gives the syntax of the language and a brief discussion about the semantics which, we hope, is intuitive for the most part. Sections 3 and 4 give the roles subtyping and recursive types have in the language. Section 5 describes multiple inheritance in our language. Finally, section 6 describes how type reconstruction works for inheritance.

# 2 Syntax

## 2.1 Overview

The language of this paper is greatly influenced by ML and by the modeling of objects done by Wand [19]. Two of the key features are the many-sorted type system, and the `class` construct. Methods to be inherited can be specified explicitly. Furthermore, there is no explicit `new` construct in the language to make new instances of classes. The whole language is summarized in Figure 1. We do not specify the syntactic category for base types, but we assume that types `unit` and `bool` are present, and we use the type `int` when convenient. We assume that the environment contains appropriate constants for these types to make the examples more appealing, in particular, we use `()` as the (only) constant of type `unit`.

| Value Bindings | Expressions |
|---|---|
| *valbind* ⇒ | *expr* ⇒ |
| *id* = *expr* | *id* |
| `rec` *id* = *function* | `self` |
| `rec` *id* = *class* | (*exprseq*) |
| *valbind* `and` *valbind* | *expr* . *id* |
| **Types** | *expr* : *type* |
| *type* ⇒ | *expr expr* |
| *basetype* | `let val` *valbind* `in` *expr* `end` |
| *typevariable* | `if` *expr* `then` *expr* `else` *expr* |
| *type* -> *type* | *function* |
| `object` { *fieldseq* } *extension* | *class* |
| `object` *typevariable* { *fieldseq* } *extension* | *function* ⇒ |
| *type* * *type* | `fn` *id* => *expr* |
| *field* ⇒ | *class* ⇒ |
| *id* : `Absent` | `class` (*idseq*) `inherits` *inherit* `methods` *valbind* `end` |
| *id* : *type* | *inherit* ⇒ |
| *extension* ⇒ | *idseq* `from` *id* `with` *expr* |
| $\epsilon$ | `all from` *id* `with` *expr* |
| *rowvariable* | *inherit* `and` *inherit* |
| *fieldseq*, *idseq*, and *exprseq* are comma separated lists of the appropriate syntactic categories | |
| `inherits` *inherit* and `methods` *valbind* are optional | |
| Only one `inherit all` clause permitted | |

Figure 1: Summary of syntax

The types in the language are somewhat unusual, involving three syntactic categories: *type*, *field*, and *extension*. The complication is caused by a special type `object`, which is the type of classes in the language. This type consists of a sequence of positive and negative information about methods in the class, optionally followed by a row variable indicating further methods are possible. Except for the treatment of recursive types, which we restrict to the type of classes, this type system is identical to the one studied by Wand [19].

In Figure 1, the syntax for types is given in the form most suitable for parsing. However, it is instructive to consider the abstract syntax of the type system, which is clearly evident in the ML data structure used by the implementation.

```
datatype
  Type =
    Basetype of Id                    |
    TypeVariable of Id                |
    Arrow of Type * Type              |
    Object of Extension               |
    RecObject of Id * Extension
and
  Field =
    Absent                            |
    Present of Type
and
  Extension =
    Empty                             |
    RowVariable of Id                 |
    Ext of Id * Field * Extension     ;
```

The form of the data structure reveals that we have a 3-sorted algebra generated by a 2-sorted variable set (there are no field variables). This is crucial to get a type reconstruction algorithm based on unification, as in ML [10]. This insight to the type reconstruction problem for object-oriented languages is originally due to Rémy [16].

In our language, identifiers are used several different ways: names for values, type variables, row variables, and method names. We have chosen disjoint classes of identifiers for each kind of use. Each class is distinguished by the initial character as specified in the following table.

| class of identifier | starting character |
|---|---|
| value variable | (alphabetic character) |
| type variable | ' |
| row variable | & |
| method names | # |

## 2.2 Example

We give a simple example to illustrate classes in the language. Defining a class parameterized by x with methods #a and #b is accomplished using the `class` construct.

```
val C = class (x) methods #a = x and #b = (fn z => x+z) end
```

This defines a class value `C` which has the following type:

```
C: int -> object{#a:int,#b:int->int}
```

The type of C is a function type, because an object is created only when the value of the x is known.

Instantiation of classes is accomplished by ordinary function application. There is no need for an explicit new construct. The following lines create two distinct instances of the class C.

```
val O  = C 0;
val O' = C 1;
```

In the first case, O.#a is equal to 0, in the second O'.#a is equal to 1.

Recursive class bindings are required, just like recursive function bindings, to construct classes that contain methods which create instances of the class when called.

## 2.3 Semantics

The computational semantics of the language is given in [1], and follows the ML implementation closely. The main point in this regard is the modeling of inheritance developed by Reddy [15] and Cook [6]. The essential idea is that a class is a higher-order function

$$\lambda x \,.\, \lambda self \,.\, R$$

where $x$ is the instance variable, *self* is the Smalltalk pseudo-variable denoting the message recipient, and $R$ is a record of methods (or class protocol).

We relate the language of this paper with that of Cardelli and Mitchell [5], where they use the elegant system summarized in the next table.

| notation | defined as | explanation |
|---|---|---|
| $\langle\rangle$ | | the empty record |
| $\langle r \mid x = a\rangle$ | | extension of record $r$, if label $x$ not present |
| $r \setminus x$ | | restriction (whether or not $x$ present) |
| $r.x$ | | field selection |
| $\langle x = e\rangle$ | $\langle\langle\rangle \mid x = e\rangle$ | definition of singleton |
| $r[x \leftarrow y]$ | $\langle r \setminus x \mid y = r.x\rangle$ | renaming |
| $\langle r \leftarrow x = a\rangle$ | $\langle r \setminus x \mid x = a\rangle$ | overriding |

The meaning of the `inherits all` construct, as in the following template,

`class` $x$ `inherits all from` $P$ `with` $y$ `methods` $m$ `=` $e$ `end`

is given by:

$$\lambda\, x \,.\, \lambda\, self \,.\, \langle (P\, y\, self) \leftarrow m = e\rangle$$

The other form of the `inherits` construct specifically names the methods:

`class` $x$ `inherits` $l$ `from` $P$ `with` $y$ `methods` $m$ `=` $e$ `end`

The method names $l$ and $m$ must be distinct. This construct is equivalent to:

$$\lambda\, x \,.\, \lambda\, self \,.\, (\langle m = e\rangle \mid l = (P\, y\, self).l)$$

# 3 Subtyping

## 3.1 Subtype relation

In a seminal paper by Cardelli on the semantics of multiple inheritance [3], the notion of record-based object-oriented languages was introduced. In that paper a subtype relation is given between record types, which captures some aspects of inheritance. In essence, that relation says that a record may be replaced by another record with additional fields without causing a type error. More formally, we write

$$\texttt{object}\ \{a_1 : \tau_1, \cdots, a_n : \tau_n\} \leq \texttt{object}\ \{b_1 : \sigma_1, \cdots, b_m : \sigma_m\}$$

if the set $\{b_1, \ldots, b_m\}$ is a subset of $\{a_1, \ldots, a_n\}$ and for all $i$ and $j$ if $a_i = b_j$ then $\tau_i \leq \sigma_j$. Although the subtype relation is not a formal part of the type system here, it is easy to order the (variable-free) types of our language by this subtype relation. This is especially helpful in measuring the amount of subtype polymorphism obtained by the type system, as not all possible replacements are allowed by the language.

## 3.2 Limitations

One problem of static type checking with subtype polymorphism is caused by the `if` statement. Consider the following expression `e`:

```
val e = if c then a else b
```

where the expression `a` is of type `object{#m:int,#l:bool}`, say, and the expression `b` is of type `object{#m:bool,#l:bool}`. By forgetting the `#m` field, a type for `e` is possible, namely `object{#l:bool}`, as it is a supertype of the type of both branches. But we cannot derive any type for the expression `e`. However, much the same effect can be achieved by explicitly constraining the types, as in the alternate definition of `e` below:

```
val e = if c then a : object {#l:bool} else b : object {#l:bool}
```

Another possibility would be to add an explicit construct to forget a method as in Cardelli and Mitchell.

Even more bizarre expressions can be type correct.

```
val e = (fn x => if c then x.#m else x)
```

This has several, in fact an infinite number, of plausible types. Without recursive types for classes it is unclear what type `e` should have. But with recursive types it does have the following type:

```
object 'a {#m : 'a} &a
```

However, it is not clear that expressions like `e` are useful. On the other hand, recursive types are necessary to obtain a reasonable notion for objects.

## 3.3 Role of `let`

Since we use parametric polymorphism to obtain subtype polymorphism, the `let` construct plays an important role, just as in ML. Consider the following expression:

```
val f = (fn v => v.#m+2)
```

We can derive that `f` has the following type:

```
object{#m:int}&a -> int
```

If expressions `a` and `b` have types

```
object{#m:int,#l:bool->bool}
object{#m:int,#k:unit}
```

then the expression

```
(fn g => g(a)+g(b))(fn v => v.#m+2)
```

cannot be typed. This is because the row variable `&a` must have `#l:bool->bool` to type the expressions `f(a)`, and the field `#l:Absent` to type `f(b)`. However, both `a` and `b` have a method `#m` of type `int->int`, which is all that function `f` requires. So the expression will not cause a run-time error.

This does not mean that there is no subtype polymorphism present in the type system. The expression `f` above can be applied to any object that has a method `#m` of type `int`, since the row variable `&a` can be instantiated to include additional methods. Just as in ML, we can write

```
let val g = (fn v => v.#m+2) in g(a)+g(b)
```

to get the function to ignore different methods at different invocations.

In ML type checking, the type variables of identifiers bound in `let` constructs are called *generic* variables [14]. Just like type variables, row variables must be either generic or non-generic. They are not all generic, since otherwise the following expression would be typable:

```
(fn x => (x.#l=true, x.#l=3))
```

This is impossible as the method `#l` cannot simultaneously be `bool` and `int`.

## 3.4 Subclasses are not subtypes

Since there is no requirement that a subclass inherit all the methods of the superclass, subclasses are not subtypes. This means that a function which operates on all instances of a superclass may not necessarily work for instances of the subclasses. For example,

```
val C = class () methods #m=3 and #l=true end
val S = class () inherits #m from C with () methods #k=4 end
```

The class `S` is *not* a subtype of the class `C`, since it does not have method `#l`. So, extracting method `#l` works on `C`, but fails on the subclass `S`. As we will see, this also has implications for typing the class construct.

# 4 Recursive types

## 4.1 Detecting errors

Indiscriminately permitting recursive types is not necessarily a good idea. For example, the expression

```
(fn x => x (x))
```

would have the type $\mu\alpha.\alpha \to \beta$. Although this is technically cute as it introduces recursion, from the practical standpoint it seems likely that many unwanted programs will be accepted by the type checker.

Recursive types are not necessary for modeling `self`, but without them a method that returns `self` is not type correct. The expression

```
class () method #m = self end
```

is not typable without recursive types. However, it is not clear that this practice should be encouraged. This idiom appears often in Smalltalk code, but only because Smalltalk methods are often executed for the side effects. A functional object-oriented language may take on an entirely different character.

## 4.2 Recursive objects

As much as we would like to avoid recursive types for reasons of simplicity, it just is not possible to model a reasonable notion of objects without them. Even a simple example like a definition of the integers requires recursive types.

```
val rec Int =
  class (x) methods #value = x and #succ = Int (self.#value+1) end
```

The method `#succ` instantiates a class, the class being defined. This requires a recursive value binding of `Int`, so that `Int` is not interpreted as a free variable, but as the class being defined. Furthermore, the method `#succ` returns the newly instantiated class, thereby requiring the recursive type:

```
object 'a {#value: int, #succ: 'a}
```

In an effort to get the best of both worlds, the language permits recursive types, but only for objects, not for arbitrary types.

### 4.3 Recursive objects and subtyping

If we have recursive types for classes, we must extend the subtype relation to them as well. This requires the introduction of notation which is helpful later in comparing the typing of inheritance with that of subtype-bounded polymorphism [7]. One can view a recursive object

`object 'a` $\{a_1$`:` $\tau_1$`, ···,` $a_n$`:` $\tau_n\}$

as a function $F[t]$ to $\{a_1 : \tau_1, \ldots, a_n : \tau_n\}$ with all occurrences of the type variable `'a` replaced by the type $t$. We view all unrollings of the types as equivalent, hence, the rule

$$\texttt{object 'a } F[\texttt{'a}] = \texttt{object } F[\texttt{object 'a } F[\texttt{'a}]]$$

This expresses the nature of recursive types as equivalent infinite trees.

Using this function notation we can give the usual rule for recursive subtyping: if, under the assumption that `'b`$\leq$`'a`, we can conclude that

$$\texttt{object } G[\texttt{'b}] \leq \texttt{object } F[\texttt{'a}]$$

then we can conclude

$$\texttt{object 'b } G[\texttt{'b}] \leq \texttt{object 'a } F[\texttt{'a}]$$

# 5 Multiple inheritance

Consider inheriting specific methods from several classes `P` and `Q`, as in the following function definition:

```
(fn P =>
   (fn Q =>
      class ()
        inherits #m, #l from P with () and #k from Q with ()
        methods  #j = 3 and #i = true
      end))
```

In our language, it is an error to specify that a class has two methods with the same name, whether the methods are inherited or not. So the problem of resolving method conflicts is avoided, even when all the other methods of `P` and `Q` are not known.

The situation changes slightly with the `inherits all` construct. For example, the subclass

```
val C = class () inherits all from P with () end
```

would have every method that is present in `P`. This is more convenient for the programmer than explicitly naming every method that `C` is to inherit from `P`. However, it is not just a notational shorthand; it leads to programs that cannot be written if all inherited methods must be explicitly named. This situation occurs when a class is passed as a parameter to a function and is used as a superclass. In the function

```
(fn P => class () inherits all from P with () methods #a=3 end)
```

it is not known what methods are in P unless the type of the actual parameter is known.

Although not evident in the syntax of the language, we do not permit more than one `inherits all` construct. We explain the problem using the following class definition:

```
val C = class () inherits all from P with () and all from Q with () end
```

where P and Q are some unspecified classes. The type of the following expression

```
(C ()).#m + 3
```

causes problems. We assume that methods from Q take precedence over methods from P. But the same problem arises regardless. Clearly the type of method #m is `int`. Therefore, one of the following statements is true about the #m method of class C:

1. Q has a method #m of type `int`, or
2. Q does not have a method #m, and P has a method #m of type `int`.

This set of possibilities cannot be represented with a single type in the type system [19]. We conjecture that the language with one `inherits all` construct has the principal type property.

In any case, the explicitly mentioned methods must take precedence over the implicit ones, as these represent the methods that the programmer is overriding from the superclass. And, as we have remarked already, all the methods explicitly specified by the program must be distinct, so explicitly mentioned methods cause no problem.

# 6 Typing inheritance

## 6.1 Abstract classes

In the modeling of objects by Wand [19], abstract classes, i.e., those deferring method definitions to subclasses, are possible. For example, the class

```
val P = class () methods #m = self.#l + 3 end
```

is an entity having the following functional type:

```
object {#l:int}&a -> object {#m:int}
```

where `object {#l:int}&a` is the type of `self`. Such a object producing function cannot be instantiated with the `new` construct. The types of `self` and the object cannot be unified, because they have different methods. But other classes can inherit from P. If those classes have an #l method of type `int`, they can be instantiated.

Currently we are not able to find a printable type for C, as we do the unification with the type of `self` when C is defined. This is in keeping with our goal of hiding the details of `self` from the programmer.

## 6.2 Typing inheritance

The problem with typing inheritance is to find a relation between subclasses and superclasses that

1. ensures `self` has the necessary methods,
2. can easily be checked, and
3. is as flexible as possible.

On the assumption that every subclass is a subtype it is possible to devise a simple type rule for the `class` construct. We give this rule below. We completely ignore class parameters, since they are not germane to the discussion on typing inheritance. ($A$ is a mapping from value identifiers to their types.)

$$\frac{A[\mathit{self} \mapsto \tau] \vdash e_i : \tau_i \qquad A \vdash P : \texttt{object}\ \alpha\ \{b_j\ :\ \sigma_j\}}{A \vdash \texttt{class inherits}\ b_j\ \texttt{from}\ P\ \texttt{methods}\ a_i\ \texttt{=}\ e_i : \tau}$$

where $\tau$ is $\texttt{object}\ \beta\ \{a_i\ :\ \tau_i,\ b_j\ :\ \sigma_j\}$ and all the methods are distinct.

It is possible to do better using subtype-bounded polymorphism [7]. The rule is as follows:

$$\frac{A[\mathit{self} \mapsto \tau] \vdash e_i : \tau_i \qquad A \vdash P : \sigma}{A \vdash \texttt{class inherits}\ b_j\ \texttt{from}\ P\ \texttt{methods}\ a_i\ \texttt{=}\ e_i : \tau}$$

where $\tau$ is $\texttt{object}\ \beta\, G[\beta]$, and $\sigma$ is $\texttt{object}\ \alpha\, F[\alpha]$, *if* the following conditions hold:

$$\rho \leq \texttt{object}\ G[\rho]\ \&\ \texttt{object}\ G[\rho] \leq \texttt{object}\ F[\rho]$$

for all types $\rho$. The first rule is a special case, as the condition holds easily.

Subtype-bounded polymorphism can type more expressions. The following is an example (from [7]):

```
val P = class () methods #i=5 and #eq = (fn o => o.#i = self.#i) end
val C = class ()
  inherits #i from P with ()
  methods #b=true and #eq = (fn o => o.#i = self.#i & o.#b=self.#b) end
```

It is unclear, however, how to arrive at the appropriate recursive types in order to apply the typing rule. This is an obstacle to finding a type reconstruction algorithm for subtype-bounded polymorphism.

One restriction with subtype-bounded polymorphism is that method overriding does not work. No types can be found for the following example,

```
val C' = class ()
  inherits #i from P with ()
  methods #b=true and #eq = 3 end
```

to make the rule applicable as the type of #eq has changed to int. Furthermore, selected inheritance makes subclasses even "farther" from subtypes.

The inherits construct in our language permits the inheritance of selected methods. This is easily implemented by code sharing, but poses special problems for type checking. Selecting some combination of methods may result in a correctly typed subclass, but others may not. For example, in the following classes

```
val P = class () methods #i = 3 and #j = self.i and k=self.i+1 end
val C = class () inherits #j from P with () methods #i = true end
```

the subclass C is type correct. But if we were to inherit both #j and #k, as in

```
val C' = class inherits #j, #k from P with () methods #i = true end
```

then the method #k no longer works.

At first it might appear necessary to keep the whole text of a class and do the type reconstruction each time methods are inherited from the class. This, fortunately, is not the case. We retain the type of self required by each individual method along with the type of the method. We can separate the typing of inherited methods from the unification incumbent in the instantiation of the class.

Now we describe the type reconstruction algorithm for the case where there is no inheritance present. For each method, we type the definition as we would type any other value binding. The type of self is introduced into the type assignment as a fresh type variable. We then store the type for self and the type of the method. After we have typed all the methods in this manner, we unify all of the types for self with the type of the class being created. This is the type the programmer sees.

Suppose the class has inherited methods. We find the types for each explicitly defined method as before. For each inherited method, we look up its type and the implicit type for self. We then complete the algorithm, as above, by unifying all the types for self and then unifying with the type of the class being created.

Consider the classes P and C above. Typing method #i in class P yields object{}&a for the type of self and int for the type of the method. Typing method #j in class P yields types object{#i:'a}&b and 'a. Typing method #k yields object{#i:int}&c and int. Thus the class being created has type object{#i:int,#j:'a,#k:int}. Unifying the three types for self with the class being created yields object{#i:int,#j:int,#k:int}.

To type C, we type #i as above yielding object{}&a and bool. We look up the information for #j from the type of P, and then we unify the types of self with the type of the class being created, object{#i:bool,#j:'a}. So C is of type object{#i:bool,#j:bool}.

# 7 Summary

We have explored the design of a statically-typed language starting from the assumption that subtype polymorphism should be obtained by parametric polymorphism. We consider the principal type property to be important and so we excluded multiple `inherit all` constructs. We wanted the language to be able to select what methods are inherited from the superclass. This leads to a model of objects in which each method is viewed as function from `self`. The types of objects become too complex in this case, so we tried to design a language which hides the types of all the implicit `self` arguments.

# References


[1] Beaven, Mike, Ryan Stansifer and Dan Wetklow. A functional language with classes. CSD-TR-946, Department of Computer Science, Purdue University, January 1990.

[2] 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*, edited by J. E. Gaffney, June 1988, pages 115–125.

[3] Cardelli, Luca. "A semantics of multiple inheritance." *Information and Computation*, volume 76, 1988, pages 138–164.

[4] Cardelli, Luca. Typeful programming. SRC Report 45, Digital Equipment Corporation, Systems Research Center, May 24, 1989.

[5] Cardelli, Luca and John C. Mitchell. Operations on records. SRC Report 48, Digital Equipment Corporation, Systems Research Center, August 25, 1989.

[6] Cook, William R. and Jens Palsberg. "A denotational semantics of inheritance and its correctness." In *Object-Oriented Programming: Systems, Languages and Applications*, edited by Norman Meyrowitz, ACM, 1989, pages 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, pages 125–135.

[8] Fuh, You-Chin and Parateek Mishra. "Type inference with subtypes." In *ESOP '88: 2nd European Symposium on Programming* edited by Harald Ganzinger, Springer-Verlag, Berlin, 1988, pages 94–130.

[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] Milner, Robin. "A theory of type polymorphism in programming." *Journal of Computer and System Science*, volume 17, number 3, 1978, pages 348–375.

[11] Milner, Robin, Mads Tofte and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[12] Ohori, Atsushi and Peter Buneman. "Static type inference for parametric classes." In *Object-Oriented Programming: Systems, Languages and Applications*, edited by Norman Meyrowitz, ACM, 1989, pages 445–456.

[13] Atsushi Ohori, Peter Buneman, and Val Breazu-Tannen. "Database programming in Machivelli—A polymorphic language with static type inference." In *Proceedings of the ACM SIGMOD Conference*, 1989, pages 46–57.

[14] Chris Reade. *Elements of Functional Programming.* Addison-Wesley, Wokingham, England, 1989.

[15] 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, pages 289–297.

[16] 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, pages 77–88.

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

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

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