Type Inference for Objects with Instance Variables and Inheritance

Mitchell Wand

February, 1989

College of Computer Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, MA 02115, USA

This Material is based on work supported by the National Science Foundation under grants DCR 8605218 and CCR 8801591.

Northeastern University College of Computer Science Technical Report NU-CCS-89-2, February, 1989. Reprinted in *Theoretical Aspects of Object-Oriented Programming* (Carl Gunter and John C. Mitchell, eds.), MIT Press, 1994, pp. 97–120.

Abstract

We show how to construct a complete type inference system for object systems with protected instance variables, publicly accessible methods, first-class classes, and single inheritance. This is done by extending Rémy's scheme for polymorphic record typing to allow potentially infinite label sets, and interpreting objects in the resulting language.

1. Introduction

Object-Oriented Programming (OOP) has become a popular paradigm because it provides a well-engineered combination of services to the programmer. The activities at work in OOP include a uniform reference principle, information-hiding, inheritance, and distributed definition. By a uniform reference principle, we refer to the use of field selection or message-passing as a generic way of retrieving data from composite structures. By information-hiding, we refer to the typical way in which the internal structure of an object is protected from the outside world: the "instance variables" of an object are manipulable only through the interface of the "methods". Inheritance refers to the manner in which new classes of objects are created as incremental variations of old ones. Distributed definition refers to the syntactic structure of class definitions in certain object systems, in which the definitions of the methods may be textually distributed through a program or system. We have listed these activities to suggest a decreasing order of unanimity: almost every OOP system provides a uniform reference principle, while systems vary widely in their treatment of inheritance and distributed definition.

In [Wand 87], we introduced a language and type system for dealing with record structures. That system addressed only the first of these activities: the use of record structures as a uniform reference principle. In this paper we show how to extend this system to model information-hiding and single inheritance. The resulting system is strongly typed: no well-typed program will ever produce a "method not found" message. The system is also polymorphic: it requires no type declarations, and procedures will work on inputs of many types whenever possible. The system is also noteworthy in that it allows classes as first-class citizens of the language.

We do this by interpreting the language of classes in the language of records. In this way, the typing rules for the language of records give strong typing for the language of classes as well.

The underlying type system for records is adapted from Rémy [Rémy 89], which gives a clean solution to the difficulties of the system in [Wand 87] and in addition allows the inference of recursive types, which are important in this application. We extend Rémy's system to allow infinite label sets, and show how the unification algorithm can be extended to handle these infinitely-wide trees.

We then establish a protocol for representing classes as records and show how class creation and invocation can be interpreted as syntactic sugar for the corresponding record operations. We also give another version in which classes are insulated from their representation as records.

We conclude with some remarks on structural and name equivalence for types, and discuss related work.

2. Records: Basic Definitions

Records are composite structures with components indexed by a fixed set L of labels. We assume that one can effectively determine whether a given label is present or absent in a record. Therefore, we model records as total functions

$$L \to (V + \{absent\})$$

For the moment, we will assume that L is finite; we will remedy this assumption in Section 4. The basic operations on records are selection, null, and extension.

- Selection along label a, written $(-).a$, selects the a-th component of the record:

$$r.a = r(a)$$

- The null record $null$ is the one with no fields: $\lambda a.\, absent$.
- Record extension is the standard extension of a function by one point:

$$(r \textbf{ with } a = v) = \lambda b.((b = a) \Rightarrow v,\ r(b))$$

We write r **with** $[a_1 = v_1; \ldots; a_k = v_k]$ as an abbreviation for r **with** $[a_1 = v_1] \ldots$ **with** $[a_k = v_k]$.

3. Strong Typing for Records

Our basic approach is to take the type of a record to be the record of the types of its components. Thus the type of a record is a function $L \to (Type + \{absent\})$. This suggests type constructors of the form:

$$\begin{aligned} \to &: Type \times Type \Rightarrow Type \\ \Pi &: [L \Rightarrow (Type + \{absent\})] \Rightarrow Type \end{aligned}$$

Using an observation of Rémy [Rémy 89], we can turn this back into an ordinary algebraic signature by introducing a new kind (called *Field*), and using the signature:

$$\begin{aligned}
\rightarrow &: \mathit{Type} \times \mathit{Type} \Rightarrow \mathit{Type}\\
\Pi &: \mathit{Field}^L \Rightarrow \mathit{Type}\\
\mathit{absent} &: \mathit{Field}\\
\mathit{pres} &: \mathit{Type} \Rightarrow \mathit{Field}
\end{aligned}$$

In this scheme, a field *absent* signifies a field which is absent from the record; a field $pres(t)$ indicates a field which is present and has a value of type t. Type schemes in which a field may be either present or absent can be modelled by using a field variable. Since the definition is inductive, semantics can be assigned to these types in an obvious way. Recursive types can be considered as well. In the final version of this system, we will allow L to be infinite, so that Π becomes an infinitary constructor. For the moment, however, it will suffice to imagine L to be finite.

In addition, we will allow type variables $\tau_1, \tau_2, \ldots$, field variables $f_1, f_2, \ldots$, and type constants (but not field constants). A *type* (or *field*) will consist of a well-kinded tree of kind *Type* or *Field*, respectively. These trees will have Π-nodes which branch $\mathrm{card}(L)$ ways, with daughter nodes indexed by L. In general, we will also allow trees to be countably deep.

Since L is finite, we will write Π as an ordinary type constructor, of arity $\mathrm{card}(L)$. In this system, we can write principal type schemes for the basic record operations:

$$\begin{aligned}
\mathit{null} &: \Pi(\mathit{absent}, \ldots, \mathit{absent})\\
(-).a &: \Pi(f_1, \ldots, \mathit{pres}(t), \ldots, f_n) \rightarrow t\\
(-)\ \mathbf{with}\ a = (-) &: \Pi(f_1, \ldots, f_n) \rightarrow t \rightarrow \Pi(f_1, \ldots, \mathit{pres}(t), \ldots, f_n)
\end{aligned}$$

Here we set $n = \mathrm{card}(L)$, the f_i are field variables, and the modified component of the Π constructors is the one corresponding to the label a.

It is instructive to analyze these typeschemes. The first says that *null* builds a record all of whose fields are absent. The second says that selection takes as input any record whose a field is present, and returns a value of the same type as that a field. The use of field variables allows this type to express the proposition that the other fields may be either present or absent. The last says that extension takes as inputs any record and any value, and returns a record of the same type as the input, except that the a field is guaranteed to be present with type t. These types are consistent with the semantics given earlier. Many other operations have types which are expressible in this language (see [Rémy 89]).

If L is finite, then this is a conventional type system (albeit with a slightly non-standard kind system), to which all the usual results on polymorphic typing apply. In particular, one can infer principal types with or without reflexive (infinitely deep) types and with or without polymorphic values created using the standard **let** construct of ML. We conjecture that other extensions, such as Mitchell's extension to subtyping on ground types [Mitchell 84, see also Wand & O'Keefe 89] or O'Toole and Gifford's quantification schemes [O'Toole & Gifford 89] are easily incorporated.

4. Dealing with Infinite Label Sets

In general it is not enough to typecheck programs with finite L. If one is checking a small module of a very large system, one may not know in advance what labels may be used in the larger system. Similar problems arise if one is incrementally checking a piece of a program in an interactive system. Hence it is necessary to provide for the infinite set of labels which are possible in the language.

When L is infinite, we will need some notation for specifying functions in Field^L, which we sometimes call *rows*. We observe that only the labels actually mentioned in the program need be dealt with explicitly; all the other labels can be treated as ellipses. Without loss of generality, let us assume that the labels which actually appear in the program are numbered 1 through n. (We will see later how this assumption may be eliminated). Let ρ, ρ', etc. be a new class of variables called *extension* variables (following [Jategaonkar & Mitchell 88]). We can think of an extension variable as a labelled ellipsis abbreviating an infinite sequence of field variables.

We write

$$\Pi[F_1, \ldots, F_n]\, \mathit{empty}$$

for the product type

$$\Pi(F_1, \ldots, F_n,\ \mathit{absent},\ \mathit{absent}, \ldots)$$

and

$$\Pi[F_1, \ldots, F_n]\rho$$

for the product type

$$\Pi(F_1, \ldots, F_n, f_{\rho,n+1}, f_{\rho,n+2}, \ldots)$$

where the $f_{\rho,i}$ are fresh field variables. We refer to the first n labels as *explicit* labels, and to the others as *implicit*.

In this way we reduce Π from an infinitary constructor to a finitary $n+1$-ary constructor, with a kind structure given by:

$$\begin{aligned}
\rightarrow &: \mathit{Type} \times \mathit{Type} \Rightarrow \mathit{Type} \\
\Pi &: \mathit{Field}^n \times \mathit{Extension} \Rightarrow \mathit{Type} \\
\mathit{absent} &: \mathit{Field} \\
\mathit{pres} &: \mathit{Type} \Rightarrow \mathit{Field} \\
\mathit{empty} &: \mathit{Extension}
\end{aligned}$$

Note that the kind *Extension* has only extension variables and the constant *empty*, denoting the ellipsis whose components are all *absent*. We then observe:

- All the constants have principal types which are finitely representable in this scheme.

- Any two finitary terms have a unifier if and only if their infinitary translations do, and their most general unifier represent the most general unifier of their translations.

Hence we can deal with unification (and principal types for the language without concatenation) by simply calculating with the representations.

While this development is adequate theoretically to deal with infinite label sets, it is inadequate to deal with the problem that led us to consider infinite label sets in the first place: namely, the problem of incrementally checking a portion of a program, without knowing the entire set of labels needed.

In order to deal with this problem, we observe that it is not necessary for all Π nodes to have exactly the same set of explicit labels. To capture these distinctions, we write a typical Π node as

$$\Pi[a_1 : F_1, \ldots a_k : F_k]\rho$$

to indicate that the explicit labels are $a_1, \ldots, a_k$.

In this language, we can succinctly write the types of the constants:

$$\begin{aligned} \mathit{null} &: \Pi[]\,\mathit{empty} \\ (-).a &: \Pi[a : \mathit{pres}(t)]\rho \to t \\ (-)\ \mathbf{with}\ a = (-) &: \Pi[a : f]\rho \to t \to \Pi[a : \mathit{pres}(t)]\rho \end{aligned}$$

The unification algorithm will work if we maintain the following invariants:

- All Π nodes with the same extension variable have the same explicit labels, so that each extension variable has a well-defined domain.
- When two Π nodes are unified, they must have the same explicit labels.

The first invariant is satisfied by the types of the constants as written above, so both invariants are true before unification.

Now, under this invariant, consider unifying two terms T_M and T_N. As we traverse these trees, we may reach corresponding Π nodes with different sets of explicit labels. In order to unify these, we first pad the nodes to give them the same set of explicit labels: let L_M be the set of labels explicit in the first node and L_N the set of labels explicit in the second node. For each label $a \in L_M \setminus L_N$, replace every node in T_M of the form

$$\Pi[a_1 : F_1, \ldots, a_n : F_n]\rho$$

by

$$\Pi[a_1 : F_1, \ldots, a_n : F_n,\ a : f_{\rho.a}]\rho$$

and each node of the form

$$\Pi[a_1 : F_1, \ldots, a_n : F_n]\,\mathit{empty}$$

by

$$\Pi[a_1 : F_1, \ldots, a_n : F_n,\ a : \mathit{absent}]\,\mathit{empty}$$

Pad T_N similarly. We can do such global padding by substituting a construction such as $[a : f_{\rho.a}]\rho_1$ for ρ, where ρ_1 is a fresh extension variable. We can then unify as usual.

Note that the creation of new variables is bounded by the number of new nodes that would be created had we done all the padding at once, by simply choosing to make all the labels in the whole program explicit before unifying. Hence the algorithm still halts, even though new variables are being introduced.

ML code for the algorithm is shown in Appendix A.

This unification algorithm can be adapted for use in a typechecker in a straightforward way. If we use an implementation which returns a substitution, then the substitution can include the substitution for padding the extension variables. This substitution can then be used to update the type hypotheses in the usual way. If we use a congruence-closure style algorithm, then the free type variables in the type hypotheses will share nodes with the variables in the types to be unified, so this updating will be done automatically by the unification algorithm.

The last difficulty to be faced in adapting the usual type inference algorithms to these infinitary trees is the treatment of **let**. The usual treatment of **let** is to create a typescheme by quantifying over all type variables not appearing in the type hypotheses, and then to create new variables for each quantified variable in the typescheme of an identifier [Clément *et al.* 86]. In our system, this might involve quantifying over an infinite number of field variables. But this is not a problem, as one can abbreviate this by quantifying over the corresponding extension variables, and generating new extension variables as needed.

This completes the description of the type inference algorithm for infinite label sets.

5. Modelling Objects

While this typing system seems to handle ordinary record structures well, one needs additional levels of detail to handle more realistic object-oriented systems. Such systems typically provide some level of information-hiding by hiding the "instance variables" of an object from the outside world, manipulating them only through publicly-available procedures called "methods."

We model an object as a record of methods. These methods are usually procedures. They share access to the instance variables that are local to the object. The instance variables are hidden from the rest of the program by scoping. A class is modelled as a procedure which takes values for the instance variables and produces an object. Thus we define

$$\begin{array}{l}\textbf{class}\ (x_1, \ldots, x_n)\\ \qquad \textbf{methods}\ a_1 = M_1; \ldots; a_k = M_k\ \textbf{end}\end{array}$$

to be syntactic sugar for the procedure

$$\lambda\ (x_1, \ldots, x_n).\, \mathit{null}\ \mathbf{with}[a_1 = M_1; \ldots; a_k = M_k]$$

which is the constructor function for this class. This constructs a record in which the method names $a_1, \ldots, a_k$ are associated with method bodies $M_1, \ldots, M_k$ respectively. Furthermore, the instance variables $x_1, \ldots, x_n$ are known to the method bodies, but are hidden from the user of this record.

In this model, we do not need an explicit **new** operator, but we will need one later for more sophisticated models; hence we define:

$$\textbf{new}\, C(N_1, \ldots, N_n) \equiv (C\ N_1 \ldots\ N_n)$$

In this model, class constructors are first-class citizens which may be named. For example, we could define:

$$
\begin{array}{l}
\textbf{let}\ \mathit{nil} = \textbf{class}\ ()\\
\qquad\qquad \textbf{methods}\\
\qquad\qquad\quad \mathit{null?} = \mathit{true};\\
\qquad\qquad\quad \mathit{car} = \mathit{error};\\
\qquad\qquad\quad \mathit{cdr} = \mathit{error}\\
\qquad\qquad \textbf{end}\\
\quad\ \ \mathit{cons} = \textbf{class}\ (a\ d)\\
\qquad\qquad\ \ \textbf{methods}\\
\qquad\qquad\qquad \mathit{null?} = \mathit{false};\\
\qquad\qquad\qquad \mathit{car} = a;\\
\qquad\qquad\qquad \mathit{cdr} = d\\
\qquad\qquad\ \ \textbf{end}
\end{array}
$$

When we expand this declaration, *nil* gets the principal typescheme

$$() \rightarrow \Pi[\mathit{null?} : \mathit{pres}(\mathrm{bool}), \mathit{car} : \mathit{pres}(\alpha), \mathit{cdr} : \mathit{pres}(\beta)]$$

(where α and β are free type variables) and *cons* gets typescheme

$$\alpha \times \beta \rightarrow \Pi[\mathit{null?} : \mathit{pres}(\mathrm{bool}), \mathit{car} : \mathit{pres}(\alpha), \mathit{cdr} : \mathit{pres}(\beta)]$$

Thus invocations of *nil* or of *cons* yield objects of the same type: a record consisting of *null?*, *car*, and *cdr* methods. The instance variables a and d are hidden except insofar as they are retrievable through the methods.

In this model, the type of an object is the type of the record of its methods. Thus the type of an object describes its behavior: it tells the names of the methods which are available for that object and the type of each method. If we define

$$
\begin{array}{l}
\textbf{let}\ \mathit{map} = \lambda f\ l.\ \textbf{if}\ l.\mathit{null?}\\
\qquad\qquad\qquad\quad \textbf{then new}\ \mathit{nil}()\\
\qquad\qquad\qquad\quad \textbf{else new}\ \mathit{cons}((f\ l.\mathit{car}),\ (\mathit{map}\ f\ l.\mathit{cdr}))
\end{array}
$$

then the principal type of *map* is

$$(\alpha \rightarrow \beta) \rightarrow \tau_\alpha \rightarrow \tau_\beta$$

where τ_a and τ_b satisfy the equations

$$\tau_a = \Pi([null? : pres(\text{bool}), car : pres(\alpha), cdr : pres(\tau_a)]\rho)$$
$$\tau_b = \Pi([null? : pres(\text{bool}), car : pres(\beta), cdr : pres(\tau_b)])$$

Thus we can deduce that *map* takes an object which behaves as a homogeneous list of α's and returns an object which behaves as a homogeneous list of β's (*cf* [Mishra & Reddy 85]).

Things get more complicated if we add **self**. To handle **self**, we modify our modelling protocol, so that a class is associated with a procedure which takes values for the instance variables and produces a procedure from objects to objects. This latter procedure takes as its input a value for **self** and produces the appropriate record of methods. Then **new** can construct an object with the correct binding for **self** by applying the fixed-point combinator to this procedure. With this protocol, we can define **class** and **new** as follows:

$$\begin{array}{l} \textbf{class } (x_1, \ldots, x_n) \\ \qquad \textbf{methods } a_1 = M_1; \ldots; a_k = M_k \textbf{ end} \\ \equiv \lambda\ (x_1, \ldots, x_n).\lambda\ \textbf{self}.\ [a_1 = M_1; \ldots; a_k = M_k] \end{array}$$

$$\textbf{new}\, C(N_1, \ldots, N_n) \equiv Y(C\ N_1 \ldots\ N_n)$$

Here we have used the fixed-point combinator on the domain of functions of records to records. This is fine for semantics but is generally not supported in applicative-order languages like Scheme or ML, which compute fixpoints only over procedures; a real implementation would need to simulate this by somewhat cleverer code.

We can obtain type rules for these constructs by expanding them into the language of records and then symbolically working out the resulting type constraints. If we do this, we get the following derived type rules for **class** and **new** :

$$\frac{A,\ \vec{x} : \vec{\alpha},\ \textbf{self} : \sigma \vdash M_j : \tau_j \quad j = 1, \ldots, k}{A \vdash \textbf{class } (\vec{x}) \textbf{ methods } \vec{a_j} = \vec{M_j} \textbf{ end} : \vec{\alpha} \to \sigma \to \Pi[\overrightarrow{\vec{a} : pres(\tau)}]}$$

$$\frac{A \vdash C : (\alpha_1 \times \ldots \times \alpha_n) \to \sigma \to \sigma \quad A \vdash N_1 : \alpha_1 \quad \ldots \quad A \vdash N_n : \alpha_n}{A \vdash \textbf{new}\, C(N_1, \ldots, N_n) : \sigma}$$

Here we have used vector notation to indicate a sequence of variables, types, or expressions.

6. Modelling Inheritance

We can now add inheritance to the model in a relatively straightforward way. We introduce the syntax

$$\begin{array}{l}\textbf{class }(x_1,\ldots,x_n)\\ \qquad \textbf{inherits } P(Q_1,\ldots,Q_p)\\ \qquad \textbf{methods } a_1 = M_1;\ldots;a_k = M_k \textbf{ end}\end{array}$$

which signifies a class which is to inherit from class P; the expressions $Q_1, \ldots, Q_p$ determine how to instantiate the instance variables of the parent class. (This protocol for initializing instance variables requires knowledge about the instance variables of the parent class; other protocols can be devised which avoid this drawback, such as that of [Reddy 88], which we discuss in Section 9).

As pointed out by Cook and others [Cook 88, Kamin 88, Reddy 88], we must be careful at this point to make sure that in any instance of this class, **self** in the methods of the parent class is bound to the *entire* object, not just the portion of the object corresponding to the parent class. Thus the parent class acts like a virtual class in Simula. This can be easily achieved through with our modelling protocol:

$$\begin{array}{l}\textbf{class }(x_1,\ldots,x_n)\\ \qquad \textbf{inherits } P(Q_1,\ldots,Q_p)\\ \qquad \textbf{methods } a_1 = M_1;\ldots;a_k = M_k \textbf{ end}\\ \\ \equiv \lambda(x_1,\ldots,x_n).\lambda\textbf{self}.\\ \qquad (P\,(Q_1\ldots Q_p)\,\textbf{self})\,\textbf{with}\,[a_1 = M_1;\ldots;a_k = M_k]\end{array}$$

Thus, when the class receives the value for **self**, it creates the record of methods for P, setting the instance variables for P to the values of Q (these may refer to **self**), and setting the value of **self** seen by the methods of P to be the **self** of the entire record. It then extends this record by adding the methods in the daughter class.

The rule for **new** remains as before:

$$\textbf{new}\,C(N_1,\ldots,N_n) \equiv Y(C\ N_1\ldots\ N_n)$$

so its typing rule is unchanged.

Expanding the definition of **class** yields the following typing rule:

$$\frac{\begin{array}{c}A,\ \vec{x}:\vec{\alpha}\vdash\vec{Q}:\vec{\beta}\\ A,\ \vec{x}:\vec{\alpha},\textbf{self}:\sigma\vdash P:\vec{\beta}\to\sigma\to\Pi[\vec{l}:\vec{F}]\rho\\ A,\ \vec{x}:\vec{\alpha},\textbf{self}:\sigma\vdash\vec{M}:\vec{\tau}\end{array}}{A\vdash\textbf{class }(\vec{x})\textbf{ inherits }P(\vec{Q})\textbf{ methods }\vec{a}=\vec{M}:\vec{\alpha}\to\sigma\to\Pi[\vec{a}:\overrightarrow{pres(\tau)}][\vec{l}:\vec{F}]\rho}$$

In the result type, we use updating notation in the metalanguage; the product type in the result is

$$\Pi(\lambda a.a = a_j \Rightarrow pres(\tau_j), a = l_i => F_i, f_{\rho.a})$$

Since we are working with a fixed set of explicit variables at any given time, this can still be expressed in the language of Section 4.

Let us consider an example which shows how this inheritance mechanism works and which also illustrates the kind of deductions which this type system can make.

$$
\begin{array}{l}
\textbf{let } A = \textbf{class } (a) \\
\qquad \textbf{methods } \mathit{sum} = \lambda().(a + \textbf{self}.b()) \textbf{ end} \\
\quad \textbf{in let } B = \textbf{class } (a,\ b) \\
\qquad\qquad\qquad \textbf{inherits } A(a) \\
\qquad\qquad\qquad \textbf{methods } b = \lambda().b \textbf{ end} \\
\qquad\quad\;\; C = \textbf{class } (b) \\
\qquad\qquad\qquad \textbf{inherits } A(5 - b) \\
\qquad\qquad\qquad \textbf{methods } b = \lambda().b \textbf{ end} \\
\qquad\quad\;\; D = \textbf{class } (b) \\
\qquad\qquad\qquad \textbf{inherits } A(5 - b) \\
\qquad\qquad\qquad \textbf{methods } c = \lambda().b \textbf{ end} \\
\qquad\quad\;\; E = \textbf{class } (a,\ b) \\
\qquad\qquad\qquad \textbf{inherits } B(a,\ b) \\
\qquad\qquad\qquad \textbf{methods } b = \lambda().5 \textbf{ end}
\end{array}
$$

Here A is a class which cannot be instantiated, since it requires a **self** containing a b method. B and C are two classes which inherit from A in different ways: B inherits by passing its second argument to A as its a instance value, so a call to *sum* on an instance of B will return the sum of a and b. C, however, passes $5 - b$ to A, so a call to *sum* on an instance of C will always return 5. The class D cannot be instantiated, since it still does not provide a b method. However, some descendant of D may eventually provide such a method.

Let us see how some of this information is reflected in the types assigned to these classes: A gets type

$$\mathrm{int} \to \Pi[b : \mathit{pres}(\mathrm{int}), \mathit{sum} : f]\rho \to \Pi[b : \mathit{absent}, \mathit{sum} : \mathit{pres}(\mathrm{int})]\rho$$

The result type for A has only a *sum* method defined, so the label b is mapped to *absent*. A's *self* argument, however must contain an integer-valued b method, so it is not self-applicable. B gets type

$$\mathrm{int} \times \beta \to \Pi[b : \mathit{pres}(\mathrm{int}), \mathit{sum} : f] \to \Pi[b : \mathit{pres}(\beta), \mathit{sum} : \mathit{pres}(\mathrm{int})]$$

inheriting its *sum* method from A. The type of B matches the type $(\alpha_1 \times \ldots \times \alpha_n) \to \sigma \to \sigma$ needed for **new**, so we can make instances of B. Note that we deduce the fact that the second argument to B is an integer only at the point that we make an instance, not at the point of class definition. This caution is necessary, as demonstrated by class E. Here the definition of the b method in B is overridden, so the second argument to B can be of any type whatsoever. The analyses of the other classes proceeds similarly.

It is significant that in this system, classes as well as objects are first-class values. They can be passed to and returned from procedures. For example, one can write

$$\lambda p.\lambda x.\, \textbf{new}\, p(x + 1)$$

This is a procedure which, given a class p, returns another procedure which, when called, creates a new instance of p with instance variable one greater than its argument. The type of this program is

$$(\text{int} \to \sigma \to \sigma) \to \text{int} \to \sigma$$

Similarly, one can write procedures which produce new classes:

$$\lambda p.\textbf{class}\ (x)\ \textbf{inherits}\ p(x+1)$$

This procedure is similar to the preceding one, except that instead of creating an instance of p, it creates a new class. This new class may then be inherited into other classes. The type of this procedure is

$$(\text{int} \to \sigma \to \Pi\rho) \to (\text{int} \to \sigma \to \Pi\rho)$$

However, passing classes as parameters has the usual problem with quantification in ML-style type systems: any polymorphism in the type of the argument is lost. Thus, it is impossible write a procedure which takes a class as an argument, and then uses that class as the parent of two different classes. There has been some progress on type inference for polymorphic parameters [Kfoury *et al.* 88], but this remains a difficult open problem.

Multiple inheritance is modellable using records, as well. For example, we could interpret

$$\begin{array}{l} \textbf{class}\ (x_1, \ldots, x_n) \\ \qquad \textbf{inherits}\ P(Q_1, \ldots, Q_p),\ P'(Q'_1, \ldots, Q'_q) \\ \qquad \textbf{methods}\ a_1 = M_1; \ldots; a_k = M_k\ \textbf{end} \end{array}$$

as

$$\begin{array}{l} \lambda(x_1, \ldots, x_n).\lambda\textbf{self}. \\ \quad (P\ (Q_1 \ldots Q_p)\ \textbf{self}) \| (P'\ (Q'_1 \ldots Q'_q)\ \textbf{self})\ \textbf{with}\ [a_1 = M_1; \ldots; a_k = M_k] \end{array}$$

where $\|$ indicates concatenation of records. Unfortunately, concatenation of records (under any reasonably polymorphic treatment of duplicate fields) does not have a principal type in our system. Consider the following example:

$$\lambda xy.(x\|y).a + 1$$

If we interpret concatenation as overwriting fields to the right, then this should be applicable to any pair of records x and y in which y has an integer a field *or* in which x has an integer a field and y has an absent a field. This may be expressed as having types of the form

$$\Pi\rho \to \Pi[a : \mathit{pres}(\text{int})]\rho' \to \text{int}$$

and

$$\Pi[a : \mathit{pres}(\text{int})]\rho \to \Pi[a : \mathit{absent}]\rho' \to \text{int}$$

This term does not have a principal type in any known system, including [Cardelli 88, Rémy 89]. It is possible, however, to extend the system we have described to generate a finite set of principal types for terms involving concatenation [Wand 89].

7. Opaque Class Systems

Our model of classes is transparent, in that the types are the same as those of the underlying de-sugared terms. It is possible to "forge" an object, since we know its representation. It may be desirable in some languages or implementations to prevent this by keeping the representation of objects hidden, and thus distinct from the representation of records. Such a distinction would be crucial to a system implemented using generic functions, for example.

A simple solution is to make objects unforgeable by using a different type constructor for records and for objects, say by adding a new type constructor

$$\mathit{Record} : \mathit{Field}^L \Rightarrow \mathit{Type}$$

with separate constructors and selectors. Thus values of object type (that is, of Π type) could be constructed only using **class** and **new**.

It would also be possible to introduce a new constructor

$$Class : \mathit{Type} \times \mathit{Type} \times \mathit{Type} \rightarrow \mathit{Type}$$

so that the type of a **class** expression would be $Class$(type of parameters, type of self, type of object). However, it is not clear what additional purpose this might serve.

8. Intensional and Extensional Types

There has been considerable confusion in the literature over the distinction between *type* and *class*. In this section, we will briefly discuss how our system addresses this issue.

Our view of *type* is structural or extensional. That is, the type of an object is a signature containing the types of its methods. A program which is type-correct in our system will never generate a "method not found" error, because selection on a label is not allowed unless that label is provably present in the record. One object is substitutable for another so long as it has the same type: such a substitution may change the behavior of the program, but it will never cause a "method not found" error.

Other systems use a name-based or intensional view of type. In such systems, each class constitutes a type, and one object is not substitutable for another unless it is of the same class. Such systems must then wrestle with the problem of making this view live compatibly with inheritance.

Why should such a restriction be seen as desirable? An attractive explanation comes from the literature on abstract data types [*e.g.* Hoare 72]. There a class is seen as some data (the representation) along with the set of permissible operations on that data. The reason that the set of operations is crucial is that there may be some invariant (the "concrete invariant") that must be maintained on the representation. For example, in a abstract data type for a histogram, one of the instance variables might always contain the total number of data points in the histogram. This requires an invariant to hold between this instance variable and the instance variable holding the histogram array.

Similarly, one might expect invariant relations to hold between different methods in an object. For example, in our previous example, B and C generate objects of the same type, but if c is an instance of C, then we expect $c.s = 5$ to hold, whereas this is not true of instances of B. If the code which uses these classes depends on such additional properties, then substituting an instance of B for an instance of C will cause the program to become incorrect.

We may hypothesize, then, that the intensional type of a class consists of the extensional or structural type that we have computed plus the invariants that are intended to hold among the instance variables and methods. One can easily sketch the semantics of such a view in a semantics of types that allows arbitrary predicates, such as the PER model [*e.g.*, Mitchell 86], but static analysis of intensional types would require full-blown theorem-proving rather than simple unification. Name equivalence may be regarded as a fail-safe approximation to typing using explicit invariants ("x is an instance of C" always implies "x satisfies C's invariants").

9. Related Work

[Cardelli 88] introduced record models of objects, including subtyping. His system did not deal with records of indefinite width, as in

$$\lambda x.x\ \mathbf{with}\ a := (x.a + 1)$$

nor did his system attempt to do type inference. [Wand 87] attempted to do type inference for records including record overwriting of this kind; unfortunately the unification algorithm in this paper was incorrect.

[Rémy 89] introduced the notion of fields, which gave an obviously correct treatment of records using the usual notion of unification, thus avoiding the complications of unification in non-free algebras. Rémy treated only finite label sets; one of the contributions of this paper is to extend this work to infinite label sets.*

The system we have used focusses on polymorphism in the procedures; Rémy also introduced another system in which the records themselves are polymorphic. This system uses the following signature:

$$\begin{aligned} \rightarrow &: \mathit{Type} \times \mathit{Type} \Rightarrow \mathit{Type} \\ \Pi &: (\mathit{Tag} \times \mathit{Type})^L \Rightarrow \mathit{Type} \\ \mathit{absent} &: \mathit{Tag} \\ \mathit{pres} &: \mathit{Tag} \end{aligned}$$

In this system, the principal type schemes of the basic operations are given by:

* An early version of [Rémy 89] remarked that infinite label sets could be treated by "lazily expanding" ellipses, but did not give an algorithm or proof.

$$null : \Pi(absent, t_1, \ldots, absent, t_n)$$
$$(-).a : \Pi(f_1, \ldots, pres, t, \ldots, f_n) \to t$$
$$(-)\ \mathbf{with}\ a = (-) : \Pi(f_1, \ldots, f_n) \to t \to \Pi(f_1, \ldots, \phi, t, \ldots, f_n)$$

In this system, a record is regarded as a polymorphic object, in which any field containing a value may be instantiated as either present (for use by selection) or as absent (forgotten). The set of terms typable under this system is incomparable with those typable under the original system. This system seems preferable for some applications, but giving it a plausible semantics remains an open problem.

[Jategaonkar & Mitchell 88] gave a type system for extendible records in ML, including ML patterns and subtyping on ground (*i.e.* name-equivalent) types. [Stansifer 88] also contributed a treatment of type inference for records.

[Reddy 88] gives a semantics for objects as closures which is very close in spirit to ours. He gives a traditional denotational semantics, whereas we give a concrete semantics [Wand 85]: a translation into an underlying lambda-calculus. This gives a direct demonstration that objects really are closures. Furthermore, again following [Wand 85], we obtain finer type information by looking at the type of the resulting terms than is possible by looking just at the denotational semantics: we derive typing rules for the source language by saying that if a source language phrase is well-typed, then its translation must be. By contrast, in a traditional denotational semantics, the translation of a source language phrase is always well-typed.

Reddy's system differs in details from ours. Most interestingly, he avoids problems with initializing instance variables by always initializing them to be new (uninitialized) L-values. The corresponding translation in our system would be something like

$$\begin{array}{l}\mathbf{class}\ (x_1, \ldots, x_n)\\ \qquad \mathbf{inherits}\ P\\ \qquad \mathbf{methods}\ a_1 = M_1; \ldots; a_k = M_k\ \mathbf{end}\\ \\ \equiv \lambda\mathbf{self}.\mathbf{letvars}\ (x_1, \ldots, x_n)\ \mathbf{in}\\ \quad (P\,\mathbf{self})\ \mathbf{with}\ [a_1 = M_1; \ldots; a_k = M_k]\end{array}$$

where **letvars** is a construct which creates and binds n new uninitialized locations. Here each class knows enough to initialize the right number of instance variables. One can do type inference for this translation as well, but one must be suitably careful about polymorphic references [*e.g.* Tofte 87]. Again, one also needs to be careful about the correct implementation of the fixed point operator.

10. Conclusions

We have presented a polymorphic type discipline for a language with objects, including hidden instance variables, methods, single inheritance, and classes as first-class values. This type discipline has decidable principal typing, obtained by interpreting classes and objects in a typed language of records. The type system for records is obtained by extending the system of Rémy by giving an algorithm for infinite label sets.

Acknowledgements

Atsushi Ohori, Luca Cardelli, Pavel Curtis, Mads Tofte, and Chris Haynes all contributed careful readings of [Wand 87]. Conversations with Ryan Stansifer, Patrick O'Keefe, Dan Friedman, John Mitchell, Lalita Jategaonkar, and Peter Wegner at various times also helped clarify the issues.

References

[Cardelli 88]
Cardelli, L. "A Semantics of Multiple Inheritance," *Information and Computation 76* (1988), 138–164.

[Clément *et al.* 86]
Clément, D., Despeyroux, J., Despeyroux, T., and Kahn, G. "A Simple Applicative Language: Mini-ML" *Proc. 1986 ACM Symp. on Lisp and Functional Programming*, 13–27.

[Cook 88]
Cook, W., "A self-ish model of inheritance," Ph.D. dissertation, Brown University, 1988.

[Hoare 72]
Hoare, C.A.R., "Proving Correctness of Data Representations," *Acta Informatica 1* (1972), 271–281.

[Jategaonkar & Mitchell 88]
Jategaonkar, L.A., and Mitchell, J.C. "ML with Extended Pattern Matching and Subtypes," *Proc. 1988 ACM Conf. on Lisp and Functional Programming*, 198–211.

[Kamin 88]
Kamin, S. "Inheritance in Smalltalk-80: A Denotational Definition," *Conf. Rec. 15th Ann. ACM Symp. on Principles of Programming Languages* (1988), 80–87.

[Kfoury *et al.* 88]
Kfoury, A.J., Tiuryn, J., and Urzycyzn, P. "A Proper Extension of ML with an Effective Type-Assignment" *Conf. Rec. 15th Ann. ACM Symp. on Principles of Programming Languages* (1988), 58–69.

[Mishra & Reddy 85]
Mishra, P., and Reddy, U.S. "Declaration-Free Type Checking" *Conf. Rec. 12th Ann. ACM Symp. on Principles of Programming Languages* (1985), 7–21.

[Mitchell 84]
Mitchell, J.C. "Coercion and Type Inference (summary)," *Conf. Rec. 11th Ann. ACM Symp. on Principles of Programming Languages* (1984), 175–185.

[Mitchell 86]
Mitchell, J.C. "A Type-Inference Approach to Reduction Properties and Semantics of Polymorphic Expressions" *Proc. 1986 ACM Conf. on Lisp and Functional Programming*, 308–319.

[O'Toole & Gifford 89]
O'Toole, J.W., and Gifford, D.K. "Type Reconstruction with First-Class Polymorphic Values," *Proc. SIGPLAN '89 Conf. on Programming Language Design and Implementation, SIGPLAN Notices 24*, 7 (July, 1989), 218–226.

[Reddy 88]
Reddy, U.S. "Objects as Closures: Abstract Semantics of Object Oriented Languages" *Proc. ACM Conf. on Lisp and Functional Programming*, (1988) 289–297.

[Rémy 89]
Rémy, D. "Typechecking records and variants in a natural extension of ML," *Conf. Rec. 16th Ann. ACM Symp. on Principles of Programming Languages* (1989), 77–88.

[Stansifer 88]
Stansifer, R. "Type Inference with Subtypes," *Conf. Rec. 15th Ann. ACM Symp. on Principles of Programming Languages* (1988), 88–97.

[Tofte 87]
Tofte, M. "Operational Semantics and Polymorphic Type Inference," Ph.D. dissertation, University of Edinburgh, 1987.

[Wand 85]
Wand, M. "Embedding Type Structure in Semantics" *Conf. Rec. 12th ACM Symp. on Principles of Prog. Lang.* (1985), 1–6.

[Wand 87]
Wand, M. "Complete Type Inference for Simple Objects" *Proc. 2nd IEEE Symposium on Logic in Computer Science* (1987), 37–44.

[Wand 89]
Wand, M. "Type Inference for Record Concatenation and Multiple Inheritance," *Information & Computation 93* (1991), 1–15. Preliminary version appeared in *Proc. 4th IEEE Symposium on Logic in Computer Science* (1989), 92–97.

[Wand & O'Keefe 89]
Wand, M., and O'Keefe, P. "On the Complexity of Type Inference with Coercion," *Conf. on Functional Programming Languages and Computer Architecture* (London, September, 1989).

Appendix A. ML Code for the Unification Algorithm

The code shown here implements the unification algorithm with recursive types and with on-demand expansion of extension variables.

```
(* Trees include Types, Fields, and Extensions.  Kinding restrictions
are not modelled. *)

datatype Tree = var of string * int |
                const of string |
```

```
                absent |
                present of Tree |
                arrow of Tree * Tree |
                record of Tree |
                extension of (string * Tree) list * Tree |
                empty ;

fun tvar s = var(s,0);

fun mapappend [] = "" |
    mapappend (x :: y) = (x ^ (mapappend y));

(* Nodes:  last component is link to representative for union/find *)

datatype Node = nullnode |
                varnode of string * int * Node ref |
                constnode of string |
                absentnode of Node ref |
                presentnode of Node * Node ref |
                arrownode of Node * Node * Node ref |
                recordnode of Node * Node ref |
                extensionnode of (string * Node) list * Node |
                              (* extension nodes are never
                              equivalenced *)
                emptynode of Node ref;

exception Link_of_Null of unit;

fun link (nullnode) = raise Link_of_Null {} |
    link (constnode(s)) = ref nullnode |
    link (varnode(s,i,l)) = l |
    link (absentnode(l)) = l |
    link (presentnode(s, l)) = l |
    link (arrownode(n1, n2, x)) = x |
    link (recordnode(_, l)) = l |
    link (extensionnode(_, _)) = ref nullnode |
    link (emptynode(l)) = l;

val gen_num = let
        val count = ref 0
        in fn x => (count := !count + 1 ; !count) end;

fun find n = if !(link n) = nullnode then n else find (!(link n));

fun equiv n n1 =
    let val r = (find n)
```

```
        and r1 = (find n1)
    in (link r) := r1 end;

(* simple tables *)

val empty_table = [];

fun add_to_table table sym v = (sym, v) :: table ;

fun lookup x ((y, v)::table) succ fail = if x = y then (succ v) else
                                             lookup x table succ fail |
    lookup x [] succ fail = fail() ;

(* translate type expression to DAG *)

fun translate (v as var(s,i)) A =
              lookup v A (fn x => (x, A))
                         (fn () => let val n = varnode (s, i, ref nullnode)
                             in (n, add_to_table A v n) end ) |
    translate (const(s)) A = (constnode(s), A) |
    translate (absent) A = (absentnode (ref nullnode), A) |
    translate (present(t)) A =
       let val (n, A) = translate t A
       in (presentnode(n, ref nullnode), A) end |
    translate (arrow(t1, t2)) A =
       let val (n1, A1) = translate t1 A ;
           val (n2, A2) = translate t2 A1
       in (arrownode(n1, n2, ref nullnode), A2) end |
    translate (record(t)) A =
       let val (n, A) = translate t A
       in (recordnode(n, ref nullnode), A) end |
    translate (extension(l,t)) A =
       let fun loop [] A = ([], A) |
               loop ((s,t)::l) A =
                 let val (n, A1) = translate t A;
                     val (l, A2) = loop l A1
                 in ((s,n)::l, A2) end;
           val (n1, A1) = translate t A ;
           val (l, A2) = loop l A1
       in (extensionnode(l, n1), A2) end |
    translate (empty) A = (emptynode(ref nullnode), A) ;

(* and back again *)

fun member x (y :: l) = if x = y then true else member x l |
    member x [] = false ;
```

```
val untranslate =
    let
        fun loop n seen =  (* assumes n is a representative *)
                if member n seen then var("...",0)
                  else case n of
                       varnode(s,i,r) => var(s,i) |
                       constnode(s) => const(s) |
                       absentnode(l) => absent |
                       presentnode(n1,l) => present(loop (find n1) (n::seen)) |
                       arrownode(n1, n2, r) =>
                          arrow(loop (find n1) (n::seen),
                                loop (find n2) (n::seen)) |
                       recordnode(t,l) => record(loop (find t)
                                                    (n::seen)) |
                       extensionnode(l,n2) =>
                          extension(map (fn (s,n1)=>(s, loop (find n1)
                                                             (n::seen)))
                                        l,
                                    loop (find n2) (n::seen)) |
                       emptynode(_) => empty |
                       nullnode => raise Link_of_Null()
        in (fn n => (loop n [])) end;

fun printn (n: Node) = print ((tree_to_string (untranslate n))^"\n");

fun printsl (sl : string list) = (print "["; (map (fn s => (print s;
    print ", ")) sl); print "]\n"; {});

(* Now we get to the interesting stuff *)

exception Kinding_Failed of Node;

(* get list of explicit labels in a row.  Assumes its argument is a
representative *)

fun explicits (extensionnode(pairs, ext)) =
       (map #1 pairs) @ (explicits (find ext)) |
    explicits (varnode(_,_,_)) = [] |
    explicits (n) = raise Kinding_Failed n ;

fun setdiff [] l2 = [] |
    setdiff (x::l1) l2 = if member x l2 then setdiff l1 l2
                                        else x::(setdiff l1 l2);

(* pad t newlabels newext -- pads row t by finding its extension
```

```
variable and then unifying it with an extension mapping newlabels to
either fresh variables or to absent.  Assumes its argument is a
       representative. Note that extension nodes are never equivalenced!
*)

fun pad t newlabels newext =
    let fun loop (extensionnode(_,t)) = loop (find t) |
            loop (n as varnode(_,_,_)) =
                 (equiv n (extensionnode
                           (map (fn s => (s, varnode("f", gen_num(),
                                                          ref nullnode)))
                                newlabels,
                            newext))) |
            loop (n as emptynode(_)) =
                 (equiv n (extensionnode
                           (map (fn s => (s, varnode("f", gen_num(),
                                                      ref nullnode)))
                                newlabels,
                            newext));
                  equiv newext (emptynode (ref nullnode))) |
            loop n = (printn n; raise Kinding_Failed n)
     in loop (find t) end;

fun find_field s (extensionnode(pairs, ext)) =
       lookup s pairs (fn f => f) (fn () => find_field s (find ext)) |
    find_field s (_) = raise Kinding_Failed (nullnode);


(* subgoals:  takes two rows, assumed to have same explicits, and
generates list of subgoals to be unified *)

(* also assumes that there are no duplicates in the explicits.  This
invariant is maintained by this portion of the code. *)

fun subgoals r1 r2 =
    let val r1 = (find r1) and r2 = find r2;
        val e1 = explicits r1
        in map (fn s => (find_field s r1, find_field s r2)) e1 end;


exception Unify_Failed of Node * Node;

fun unify n1 n2 =
 let val n1 = find n1 and n2 = find n2 in
     if n1 = n2 then {}
     else case (n1, n2) of
```

```
        (varnode(_,_,_), n2) => equiv n1 n2 |
        (constnode(s), constnode(t)) => if s = t then {} else raise
                                               Unify_Failed (n1, n2) |
        (n1, varnode(_,_,_)) => equiv n2 n1 |
        (absentnode(_) ,absentnode(_)) => equiv n1 n2 |
        (presentnode(t1,_),presentnode(t2,_)) =>
           (equiv n1 n2; unify t1 t2) |
        (arrownode(t1, t2, r), arrownode(u1, u2, s)) =>
           (equiv n1 n2; unify t1 u1; unify t2 u2) |
        (recordnode(t1,_),recordnode(t2,_)) =>
           (equiv n1 n2; unify_row t1 t2) |
        (extensionnode(_,_), _) => (printn n1; raise Kinding_Failed n1) |
        (_, extensionnode(_,_)) => (printn n2; raise Kinding_Failed n2 )|
        n => raise Unify_Failed (n1, n2)
    end
and unify_row t1 t2 =
      let val exp1 = explicits t1 and exp2 = explicits t2;
          val a = setdiff exp1 exp2 and b = setdiff exp2 exp1
          val r = varnode("r",gen_num(), ref nullnode);
      in (pad t1 b r; pad t2 a r;
          map (fn (n1,n2) => unify n1 n2) (subgoals t1 t2);
          {}) end;


fun unify1 n1 n2 = unify n1 n2
           handle Unify_Failed (n1, n2) => (print "Unify Failed:\n";
                                            printn n1; printn n2; ()) |
                  Kinding_Failed (n) => (print "Kinding Failed:\n";
                                         printn n; ()) ;

fun testit () =
        let val (n1, A) =
                translate (arrow(tvar "x", arrow(tvar "y", tvar "y"))) [];
            val (n2, A) = translate (arrow(tvar "x", tvar "x")) A
        in (unify n1 n2;
            print (tree_to_string (untranslate n1));
            print "\n"; true) end;

fun test2 () =
    let val (n1, A) = translate (arrow(tvar "x", tvar "y")) [];
        val (n2, A) = translate (tvar "x") A
    in (unify n1 n2;
        print (tree_to_string (untranslate n1));
        print "\n"; true) end;

fun make2 () = let
```

```
    val (n1, A) = translate
                      (record
                       (extension([("a",present(const "bool"))],
                                  tvar "r1"))) [];
    val (n2, A) = translate
                      (record
                       (extension([("b",present(const "int"))],
                                  tvar "r2"))) A
    in (n1, n2) end;



fun test3 () = let val (n1, n2) = make2 () in
 (unify1 n1 n2;
       print (tree_to_string (untranslate n1)) ;
       print "\n"; true) end;
```