Type Inference for Record Concatenation and Multiple Inheritance

Mitchell Wand

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 numbered DCR-8605218 and CCR-8801591.

A preliminary version of this paper appeared in *Proc. 4th Ann. Symp. on Logic in Computer Science* (1989)

Proposed Running Head: **Type Inference for Record Concatenation**

Abstract

We show that the type inference problem for a lambda calculus with records, including a record concatenation operator, is decidable. We show that this calculus does not have principal types, but does have finite complete sets of types: that is, for any term M in the calculus, there exists an effectively generable finite set of type schemes such that every typing for M is an instance of one the schemes in the set.

We show how a simple model of object-oriented programming, including hidden instance variables and multiple inheritance, may be coded in this calculus. We conclude that type inference is decidable for object-oriented programs, even with multiple inheritance and classes as first-class values.

1. Introduction

A practical motivation for type inference is to ensure an operational safety property of programs that are well-typed: that is, when we execute a well-typed program, we are guaranteed that we will never get an error message such as "bad function `nil`."

Our goal is to extend this safety property to programs involving records and objects. Here the safety property is that we will never get a message such as "can't find field" when we attempt to do a field extraction operation. For object-oriented programming, we wish to guarantee that we will never get messages like "can't find method."

We begin by reviewing, in Section 2, the basic operations on records. In Section 3, we show how objects and classes can be modelled as syntactic sugar for record operations. In this way, typing results for records can be used for object-oriented programs. We then consider the type inference problem for the lambda-calculus with records. These properties differ dramatically depending on the record constructors considered. In Section 4, we review Rémy's solution for type inference when the constructor is extension by a single field (record `cons`). In Section 5, we show how this system can be extended to record concatenation (record `append`). In Sections 6 and 7, we show how this approach can be extended to handle unbounded sets of labels. Sections 8 and 9 discuss related work and conclusions.

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

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

For the moment, we will assume that L is finite; we will relax this assumption in Section 6.

The basic operations on records are selection, null, extension, and concatenation.

- 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:

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

  We write $r \,\mathbf{with}\, [a_1 = v_1; \ldots; a_k = v_k]$ as an abbreviation for $r \,\mathbf{with}\, [a_1 = v_1] \,\mathbf{with}\ \ldots \mathbf{with}\, [a_k = v_k]$.
- Record concatenation is the standard union of two partial functions:

$$(r \,\|\, r') = \lambda a.(\text{inleft?}(r'(a)) \Rightarrow r'(a),\ r(a))$$

Concatenation and extension both overwrite to the right: the rightmost field which is present is the one which appears in the answer.

Concatenation poses severe problems for typing systems. Consider the term

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

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 term does not have a principal type in any known system, including [Rémy 89, Cardelli 88]. We shall show that its types are generated by two type schemes.

3. Objects

Our main practical motivation in considering records is that we can model objects and classes using these operations. We model an *object* as a record of *methods*. These methods are usually procedures. They share access to a set of *instance variables* that are local to the object. The instance variables are hidden from the rest of the program by scoping. Furthermore, the methods may refer to the object itself through the identifier **self**. A *class* is modelled as a procedure which takes values for the instance variables and an object (the **self**) and produces an object. With these conventions, we can think of class definition and instantiation as syntactic sugar for the following record operations:

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

$$\textbf{make-instance}\, C(N_1, \ldots, N_n) \equiv Y(C\ N_1 \ldots\ N_n)$$

Here the body of the class definition builds up a record of methods by starting with the empty record and adding methods one at a time. The **make-instance** operator uses the fixed-point operator Y to guarantee that **self** is bound to the whole object.

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

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

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.

As pointed out by Cook and others [Cook 87, 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 using the same protocol we have been using:

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

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 one at a time. The **make-instance** operator remains as before.

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

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

as

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

In this way, we treat an object-oriented program as syntactic sugar for a term in the lambda-calculus with records. An unusual feature of this language is that classes are ordinary data values which can be passed as parameters. Thus one could write a class transformer:

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

This translation enables us to make the connection between record concatenation and multiple inheritance. If we have a type system for the lambda-calculus with records, then we can decide typing for programs in the object-oriented language, simply by expanding the syntactic sugar. Results such as subject reduction, principal types, and semantics are similarly inherited. Some of the details are worked out in [Wand 89]. We proceed, therefore, to consider the type inference problem for the lambda calculus with records.

4. Type Inference 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 $L \to (\mathit{Type} + \{\mathit{absent}\})$. This suggests type constructors of the form:

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

Rémy [Rémy 89] observed that this may be turned back into an ordinary algebraic signature by introducing a new kind (which he called a *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. Schemas 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.

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 schema. 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.

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] or O'Toole and Gifford's quantification schemes [O'Toole & Gifford 89] are easily incorporated.

This gives a solution for the case of record extension (see [Rémy 89] for some variants). We next turn to the more difficult case of type inference for concatenation.

5. Dealing with Concatenation

Unfortunately, in this system it is impossible to assign a principal type to the concatenation operator. For example, let us show that $\lambda xy.((x \,\|\, y).a + 1)$ has no principal type in this system. Let L be $\{a\}$. Then this term should have type

$$\Pi(absent) \rightarrow \Pi(pres(\text{int})) \rightarrow \text{int}$$

and it should also have type

$$\Pi(pres(\text{int})) \rightarrow \Pi(absent) \rightarrow \text{int}$$

Therefore, if it had a principal type, that type must be at least as general as

$$\Pi(f_1) \rightarrow \Pi(f_2) \rightarrow \text{int}$$

This is not a reasonable type for this term. Therefore concatenation has no principal type which satisfies these minimal expectations.

In order to analyze concatenation, we need to look more closely at the type assignment rules for the lambda calculus.

It is useful to think of the ordinary type inference rules (in the absence of **let**) as a set of constraints on the type expressions which appear in the derivation. In this view, we assign a type variable to every subterm and to every binding occurrence of a variable. The type inference rules may be stated as constraints on the types which can appear in the corresponding positions in the derivation. We write a constraint for each node in the parse tree (isomorphic, of course, to the derivation tree):

- for each applied occurrence of a variable x, generate the constraint $t_x = A(x)$, where t_x is the type variable corresponding to this applied occurrence of x and $A(x)$ is the type variable corresponding to the relevant binding occurrence of x.
- for each occurrence of an application $(M\ N)$ generate the constraint $t_M = t_N \rightarrow t_{(M\ N)}$, where each type variable is the type variable corresponding to the occurrence of the indicated term.
- for each occurrence of an abstraction $\lambda x.M$, generate the constraint $t_{(\lambda x.M)} = t_x \rightarrow t_M$, where each type variable is the type variable corresponding to the indicated occurrence (a binding occurrence in the case of t_x).

It is easy to see that this formulation is equivalent to the usual inference rules, so that the solutions to the generated set of equations correspond to the possible type derivations. Thus the existence of most general unifiers implies the existence of principal types. This reduction is folkloric [*e.g.* Cardelli 85, Clément *et al.* 86], and is implicit in [Hindley 69, Milner 78].

It is not possible to state a typing rule for concatenation as an equation in this style, since concatenation has no principal type, but it is possible to express a sound typing rule for concatenation using a *disjunction* of equations:

- For each occurrence of a concatenation $(M \,\|\, N)$, generate the following constraints:

$$
\begin{aligned}
t_M &= \Pi(f_1, \ldots, f_n) \\
t_N &= \Pi(g_1, \ldots, g_n) \\
t_{(M \,\|\, N)} &= \Pi(h_1, \ldots, h_n) \\
(g_i = \mathit{pres}(t_i) \wedge h_i = g_i) \vee (g_i &= \mathit{absent} \wedge\ h_i = f_i) \quad i = 1, \ldots, n
\end{aligned}
$$

These constraints reflect the following analysis: all of M, N, and $M \,\|\, N$ must be records, of some yet-to-be-determined composition. For each field, either the field is present in N, in which case the field in N is present in the result, or else the field is absent in N, in which case the field in the result is the same as it is in M, whether it be present or absent.

These constraints determine a type inference rule: that is, they form an acceptance criterion on a type derivation tree. Writing out the rule in the usual deduction-rule form is left as a tedious exercise for the reader.

The constraints can also be used for type reconstruction. We no longer have a conjunction of equations, but we have a positive boolean combination of equations. Hence we can expand it into disjunctive normal form, getting a disjunction of conjunctions of equations. Each conjunction can be analyzed to get a most general unifier, yielding a finite set of types whose substitution instances are precisely the typings of the original term. This proves the main theorem:

Theorem. *Given a closed term M, we can effectively determine whether M has a type. In particular, we can generate a finite set of type schemes such that the types of M are exactly the substitution instances of these schemes.*

(Here, for convenience, we have stated the result for closed terms; the result for terms with free variables is slightly harder to state but no more difficult.)

For our motivating example $\lambda xy.((x \,\|\, y).a + 1)$, it is easy to see that a complete set of types is

$$\Pi(f_1, \ldots, \mathit{pres}(\mathrm{int}), \ldots, f_n) \to \Pi(g_1, \ldots, \mathit{absent}, \ldots, g_n) \to \mathrm{int}$$

and

$$\Pi(f_1, \ldots, f_n) \to \Pi(g_1, \ldots, \mathit{pres}(\mathrm{int}), \ldots, g_n) \to \mathrm{int}$$

where, as usual, the expanded argument to Π is the one corresponding to the a field. From this it is also easy to see that this term has no principal type, as any type which has both these types as instances also has instances which are not legitimate types for this term.

The number of types generated can be large of course: it may be as large as 2^{kn}, where $n = \mathrm{card}(L)$ and k is the number of occurrences of concatenation in the program. In practice, one would attempt to solve the equations as much as possible before expanding the disjunctions, and to prune unsatisfiable disjunctions as quickly as possible. This is a reflection of a real difficulty in object-oriented programming systems: systems with multiple inheritance go to great lengths to determine from which ancestor a particular method is inherited.

6. 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*. Let us assume without loss of generality that the labels which actually appear in the program are numbered 1 through n, and let ρ, ρ', etc. be a new class of variables called *extension* variables. We write

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

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*. We can think of an extension variable as a labelled ellipsis.

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 that all the constants have principal types which are finitely representable in this scheme:

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

Furthermore, we can calculate most general unifiers of infinitary terms using the finitary representations:

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

Proof: Let Φ be the map from finitary to infinitary terms defined above. Then, we observe that a finitary term u is an instance of a finitary term t if and only if $\Phi(u)$ is an instance of $\Phi(t)$. Therefore, if finitary terms t_1 and t_2 are unifiable, then $\Phi(t_1)$ and $\Phi(t_2)$ are unifiable, since $\Phi(\mathrm{mgu}(t_1, t_2))$ is a common instance.

To get the converse, we show that if $\Phi(t_1)$ and $\Phi(t_2)$ are unifiable, then their (infinitary) mgu is of the form $\Phi(u)$. To show this, we proceed by induction on the size of t_1 and t_2. If either t_1 or t_2 is of the form $\Pi[F_1, \ldots, F_n]\, \mathit{empty}$, then the most general unifier of $\Phi(t_1)$ and $\Phi(t_2)$ is of the form $\Phi(\Pi[F_1', \ldots, F_n']\, \mathit{empty})$. The only other interesting case is when $t_1 = \Pi[F_1, \ldots, F_n]\rho$ and $t_2 = \Pi[F_1', \ldots, F_n']\rho'$. Then we have

$$\begin{aligned}\Phi(t_1) &= \Pi(F_1, \ldots F_n, f_{\rho,n+1}, \ldots) \\ \Phi(t_2) &= \Pi(F_1', \ldots F_n', f_{\rho',n+1}, \ldots)\end{aligned}$$

If these have a most general unifier, then $f_{\rho,n+i}$ and $f_{\rho',n+i}$ must have a common instance. This can only be the field constant *absent* or a variable. Because each such variable appears only in the $n+i$-th position of some Π node, they can unify only with other $n+i$-th variables. Hence either the ρ variable unifies with an *empty*, and all the variables unify to *absent*, or else they all unify to variables. In either case, the most general unfifier is representable as $\Phi(u)$ for some finitary term u.

Furthermore, by our observation above, u must be the most general unifier of t_1 and t_2. *QED*

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

We next consider how to deal with concatenation in the presence of infinite L. We cannot directly extend the version for finite L because it would require generating infinitely many disjunctions. Instead, let us define an *extension constraint* to be a formula of the form

$$\rho_1 \,\|\, \rho_2 = \rho_3$$

An extension constraint abbreviates the infinite set of disjunctions

$$\begin{aligned}&(f_{\rho_2,n+i} = \mathit{pres}(t_{n+i}) \wedge\ f_{\rho_3,n+i} = f_{\rho_2,n+i}) \\ &\quad \vee\ (f_{\rho_2,n+i} = \mathit{absent} \wedge f_{\rho_3,n+i} = f_{\rho_1,n+i})\end{aligned}$$

for $i > 0$. We say that a substitution *satisfies* a set of extension constraints iff it assigns types and fields to all the type and field variables to make each of these disjunctions true. Note also that every set of extension constraints is satisfiable: just set all the ρ_i to *empty*.

Now we can state the rules for generating the constraints. We generate constraints for the ordinary terms as before. The rule for concatenation is:

- For each occurrence of a concatenation $(M \,\|\, N)$, generate the following constraints:

$$\begin{aligned}
t_M &= \Pi[f_1 \ldots, f_n]\rho_1 \\
t_N &= \Pi[g_1, \ldots, g_n]\rho_2 \\
t_{(M \,\|\, N)} &= \Pi[h_1, \ldots, h_n]\rho_3 \\
(g_i = \mathit{pres}(t_i) \wedge h_i = g_i) \vee (g_i &= \mathit{absent} \wedge h_i = f_i) \quad i = 1, \ldots, n \\
(\rho_1 \,\|\, \rho_2) &= \rho_3
\end{aligned}$$

We can expand into disjunctive normal form again, to get a disjunction of formulas of the form $(E \wedge C)$, where E is a set of equations and C is a set of extension constraints. We can then unify each disjunct individually to get a most general unifier and a set of row constraints.

In doing the unification, substitutions for extension variables are of course performed on C as well. The row constraints can also be simplified using the rules

$$\begin{aligned}
(\mathit{empty} \,\|\, \rho) &= \rho \\
(\rho \,\|\, \mathit{empty}) &= \rho
\end{aligned}$$

but this is not necessary to obtain the result.

This gives us our main theorem, which again we state just for the case of closed terms:

Theorem. *Given a closed term M, we can effectively generate a finite set $\mathcal{S}$ such that*

(1) $\mathcal{S}$ consists of pairs (C, T) such that C is a set of extension constraints and T is a type scheme, and

(2) T' is a type for M iff only there is a pair (C, T) in $\mathcal{S}$ and a substitution σ such that σ satisfies C and $T' = T\sigma$.

Corollary. *Given a closed term M, we can effectively determine whether M has a type.*

Proof: Generate a set of pairs as above. If the set is empty, then M has no type. If the set is non-empty, choose one pair, and substitute *empty* for all the extension variables. This gives a type for M. *QED*

7. Dealing Better with Infinite Label Sets

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 types to have exactly the same set of explicit labels. 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 as follows:

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

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.
- When two extension variables appear in an extension constraint, they must have the same explicit labels, so that their domains are the same.

The first invariant is satisfied by the types of the constants as written above.

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 $a \in L_N \setminus L_M$, 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

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

by

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

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.

By construction, extension constraints always start off with all of their variables having the same explicit labels (in fact, they will start with the set of explicit variables being empty); as substitution affects these constraints, we must pad the extension constraints as well.

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.

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.

8. 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. The inability of this system, and even of the more powerful system Bounded Fun [Cardelli & Wegner 85] to deal with this record updating problem has been a topic of recent discussion on the Types electronic mailing list [Meyer 88].

The language used in this paper, which is capable of dealing with record overwriting of this kind, was introduced in [Wand 87], which also attempted to do type inference for this language; unfortunately the unification algorithm in that paper was incorrect. [Rémy 89] introduced the notion of fields, which gave an obviously correct treatment of records using the usual notion of unification.

The system we have used focuses on polymorphism in the procedures; Rémy also introduced another system in which the records themselves are polymorphic. 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] give a type system for extendible records in ML, including ML patterns and subtyping on ground (*i.e.* name-equivalent) types. We conjecture that our system can replace the cut-restrictions in their system, and that the resulting systems will fit together nicely. [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. By looking at the type of the resulting terms, we obtain finer type information than is possible by looking just at the denotational semantics. We then derive typing rules for the source language by saying that if a source language phrase is well-typed, then its translation must be.

9. Conclusions

We have presented an algorithm for doing type inference for the lambda calculus with records and record concatenation. By treating object-oriented programs as syntactic sugar for terms in this language, this system enables us to do ML-style type inference for object-oriented programs with multiple inheritance, even including classes as first-class data objects.

Acknowledgements

Conversations with Patrick O'Keefe, Denis Kfoury, and Peter Wegner were helpful in understanding the issues and refining the presentation.

References

[Cardelli 85]
Cardelli, L. "Basic Polymorphic Typechecking," *Polymorphism Newsletter 2*,1 (Jan, 1985). Also appeared as Computing Science Tech. Rep. 119, AT&T Bell Laboratories, Murray Hill, NJ.

[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.

[Cardelli & Wegner 85]
Cardelli, L., and Wegner, P. "On Understanding Types, Data Abstraction, and Polymorphism," *Computing Surveys 17* (1985), 471–522.

[Cook 87]
Cook, W. "A self-ish model of inheritance," manuscript, 1987.

[Hindley 69]
Hindley, R. "The Principal Type-Scheme of an Object in Combinatory Logic," *Trans. Am. Math. Soc. 146* (1969) 29–60.

[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.

[Meyer 88]
Meyer, A. (moderator), types electronic mailing list, `types @ theory.lcs.mit.edu`

[Milner 78]
Milner, R. "A Theory of Type Polymorphism in Programming," *J. Comp. & Sys. Sci. 17* (1978), 348–375.

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

[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. "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.

[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 Objects with Instance Variables and Inheritance," Northeastern University College of Computer Science Technical Report NU-CCS-89-2, February, 1989.