Complete Type Inference for Simple Objects

Mitchell Wand

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

Abstract

We consider the problem of strong typing for a model of object-oriented programming systems. These systems permit values which are records of other values, and in which fields inside these records are retrieved by name. We propose a type system which allows us to classify these kinds of values and to classify programs by the type of their result, as is usual in strongly-typed programming languages. Our type system has two important properties: it admits multiple inheritance, and it has a syntactically complete type inference system.

1. Introduction

An important characteristic of object-oriented systems is *inheritance*, in which a type t_1 is said to *inherit* from type t_0 if it is an extension of t_0, that is, if it has all the fields of t_0, plus perhaps some others. Any function that takes an argument of type t_0 should be able to take an argument of type t_1 as well.

We think of objects as records with named fields. This gives a *structural* treatment of inheritance: a function which expects a record with certain fields should be applicable to *any* record containing those fields. For example, consider the following pseudocode.

```
type movable-object = [mass: real, position: point,
                       velocity: real];
     car = [id-number: string, mass: real,
            position: point, velocity: real];
     submarine = [id-number: integer,
                  available-missiles: integer,
                  owner: country, captain: person,
                  mass: real,
                  position: point, velocity: real];
     weapons-system = [id-number: integer,
                       available-missiles: integer,
                       position: point];

function momentum (x) = x.mass*x.velocity;
```

This Material is based on work supported by the National Science Foundation under grant numbers MCS 8303325, MCS 8304567, and DCR 8605218.

The function momentum should be applicable to both cars and submarines. We can think of cars and submarines as inheriting from movable objects. This model also permits multiple inheritance: a submarine is both a movable object and a weapons system, because any function applicable to a weapons system will be applicable to a submarine.

Cardelli [Cardelli 84] has proposed a type system (which we call C84) that accounts for inheritance of this sort. He proved the soundness of a semantics for this system. Unfortunately, C84 sacrifices a useful property of the simply-typed lambda-calculus (as exemplified by the ML system [Gordon *et al.* 78]): the solvability of the type *inference* problem. That is, we would like to write our programs without having to include any type information, and then let the compiling system infer the types of all the bound variables.

In this paper, we present a variant of the simply-typed lambda calculus which handles simple objects, that is, records with named fields. Following Cardelli, it also handles variants with named alternatives. Our version includes some minor variations on C84, which allow parameterized type inheritance and which permit a more obvious semantics.

We then give a complete type inference algorithm for this system. That is, we show that it is decidable whether a term has a type in our system, and we furthermore generate the principal type of a term. The method for proving the completeness of the type inference algorithm may be generalized to other type inference problems.

We close with some remarks about the differences between our system and Cardelli's, and on the applicability of this type-checking scheme to practical object-oriented programming systems.

2. The Type System

Records with named fields are a generalization of pairs. The generalization of the simply typed lambda-calculus to pairs and binary unions is easy, and is shown in Figure 1. For product types, we have a pair operator and two projection operators. For union types, we have two injection operators (which conceptually "add a tag bit") and a selection operator. The selection operator takes an element of $\tau_1 + \tau_2$ and two functions, one of type $\tau_1 \to \tau_3$ and one of type $\tau_2 \to \tau_3$. It examines the first argument, strips off the "tag bit", and applies one or the other function to the underlying datum. This can be expressed by the axioms

$$\mathbf{case}(\mathbf{inL}\, x) f g = f x$$
$$\mathbf{case}(\mathbf{inR}\, y) f g = g y$$

$$\langle \text{type} \rangle ::= \langle \text{base type} \rangle \mid \langle \text{type} \rangle \to \langle \text{type} \rangle \mid \langle \text{type} \rangle \times \langle \text{type} \rangle \mid \langle \text{type} \rangle + \langle \text{type} \rangle$$

$$\frac{A \vdash M_1 : \tau_1 \quad A \vdash M_2 : \tau_2}{A \vdash \mathbf{pair}\, M_1 M_2 : \tau_1 \times \tau_2}$$

$$\frac{A \vdash M : \tau_1 \times \tau_2}{A \vdash \mathbf{lson}\, M : \tau_1}$$

$$\frac{A \vdash M : \tau_1 \times \tau_2}{A \vdash \mathbf{rson}\, M : \tau_2}$$

$$\frac{A \vdash M : \tau_1}{A \vdash \mathbf{inL}\, M : \tau_1 + \tau_2}$$

$$\frac{A \vdash M : \tau_2}{A \vdash \mathbf{inR}\, M : \tau_1 + \tau_2}$$

$$\frac{A \vdash M : \tau_1 + \tau_2 \quad A \vdash N : \tau_1 \to \tau_3 \quad A \vdash P : \tau_2 \to \tau_3}{A \vdash \mathbf{case}\, M N P : \tau_3}$$

Figure 1. Rules for binary products and unions

It is easy to show, using the techniques sketched in Section 4, that the principal typing problem for this system is solvable. In fact, most existing implementations of simply typed lambda-calculus (eg ML [Gordon *et al.* 78], SPS [Wand 84]) tacitly make this extension.

We extend the rules for binary unions and products to arbitrary labelled unions and products as follows:

Let L be a fixed countable set of *labels* $a_1, a_2, \ldots$. If $\rho : D \to X$, where D is a finite subset of L, (that is a family of elements of X indexed by a finite subset of L) then we call ρ a *row* of X's. (This terminology is stolen from Algol 68). If we fix an ordering on L, then any row has a canonical finite representation $\langle (a_{i_1}, x_1), \ldots, (a_{i_n}, x_n) \rangle$, where the a_{i_k} are in increasing order.

In this paper, we use the word "type" to refer to certain syntactic objects, which may be thought of as expressions or finite trees in which rows are replaced by their canonical representations.

Definition. The set of types T is the smallest set of expressions such that:

(1) T contains the base types.

(2) T is closed under the binary operation $\to$.

(3) If ρ is a row of types, then $\Sigma\rho$ and $\Pi\rho$ are in T.

It is easy to give semantics to these syntactic objects. Given an assignment of sets to the base types, we interpret $\to$ as the function space (as usual), we interpret $\Sigma\rho$ as the disjoint union $\sum_{a \in \mathrm{dom}\,\rho} \rho(a)$, and we interpret $\Pi\rho$ as the product $\prod_{a \in \mathrm{dom}\,\rho} \rho(a)$.

Let $\rho[a \leftarrow \tau]$ denote the usual extension operation on partial functions.

We can now give the syntactic constructions dealing with records and state their typing rules:

$$\emptyset \vdash \mathbf{null} : \Pi\emptyset$$

$$\frac{A \vdash M : \Pi\rho \quad A \vdash N : \tau}{A \vdash (M\ \mathbf{with}\ a := N) : \Pi(\rho[a \leftarrow \tau])}$$

$$\frac{A \vdash M : \Pi\rho \quad a \in \mathrm{dom}\,\rho}{A \vdash M.a : \rho a}$$

The first rule creates a record with no fields. The second rule allows us to extend an arbitrary record by adding a new field or changing the value of an existing field. (By contrast, C84 allows only the creation of records of fixed length. This is like only having the primitive `list` instead of having `cons` and `nil`.) The last rule provides selection in the obvious way.

We may write down the operations for labelled unions similarly:

$$\frac{A \vdash M : \tau}{A \vdash \mathbf{inj}\, a\, M : \Sigma\rho[a \leftarrow \tau]}$$

$$\frac{A \vdash M : \Sigma\rho \quad a \in \mathrm{dom}\,\rho \quad A \vdash N : \rho(a) \to \tau \quad A \vdash P : \tau}{A \vdash \mathbf{ifcase}\, a M N P : \tau}$$

These rules are simply extensions of the corresponding rule for binary unions. The first allows injection of a quantity into a arbitrary labelled union. The second selects one case of a labelled union: if the value of M arises from an injection along label a then N is applied to the value; otherwise the value of the entire expression is that of P. The intended semantics may be captured by the axioms:

$$\mathbf{ifcase}\, a\, (\mathbf{inj}\, a\, v) f y = f v$$
$$\mathbf{ifcase}\, a\, (\mathbf{inj}\, b\, v) f y = y \quad (b \neq a)$$

The typing rules are obviously sound with respect to the coproduct semantics for Π. We thus avoid the rule for $[M$ as in C84, which causes the only semantic difficulties in that system

3. The Type Inference Problem

The type inference problem may be stated as follows:
Given a term M, for which A and τ do we have $A \vdash M:\tau$?

Following [Cardelli 85] and [Milner 78], we solve this problem by reducing it to a unification problem. We do this by using type expressions to keep track of the possible types that might occur at each node of the derivation tree. We then generate equations between the type expressions which will ensure that the tree is a legal derivation tree. Then the solutions to the equations are in correspondence with all the possible derivation trees. To do this, we must first state the appropriate notions of terms, equations, and solutions.

Since we need to represent both types and rows of types, we define type-expressions ($\langle$TE$\rangle$) and row-expressions ($\langle$RE$\rangle$).

$$\langle \text{TE} \rangle ::= \langle \text{base type} \rangle \mid \langle \text{type variable} \rangle \mid \langle \text{TE} \rangle \to \langle \text{TE} \rangle \mid \Sigma\langle \text{RE} \rangle \mid \Pi\langle \text{RE} \rangle$$
$$\langle \text{RE} \rangle ::= \textbf{emptyrow} \mid \langle \text{row variable} \rangle \mid \langle \text{RE} \rangle[\langle \text{label} \rangle \leftarrow \langle \text{TE} \rangle]$$

In the last production, $(*[* \leftarrow *])$ is just a 3-place syntactic operation, which we write using infix notation.

Define a *model* to be a map from type- and row- variables to types and rows, respectively. Given a model μ, we can interpret type- and row-expressions, as types and rows, interpreting $\to$, Σ, and Π in the obvious way, **emptyrow** as the empty row, and interpreting $(*[* \leftarrow *])$ as function extension. This gives us a meaning for $\mu \models E$, where E is a set of equations between type- and row-expressions. (We call these just sets of equations, for short).

We also need to consider *substitutions*, which are maps from type- and row- variables to type- and row-expressions. A substitution is called a *ground* substitution if its range consists of expressions without variables. Clearly each ground substitution determines a model, and each model determines a ground substitution obtained by using the canonical representation for each row. We will therefore use models and ground substitutions interchangeably below.

Composition of substitutions is defined in the usual way: $\sigma \circ \tau$ has domain $\text{dom}\,\sigma \cup \text{dom}\,\tau$, and $(\sigma \circ \tau)(v)$ is $\tau(v)$ for $v \notin \text{dom}\,\sigma$, and $(v\sigma)\tau$ otherwise.

Lemma *(Substitution Lemma): Let E be a set of equations, μ a model, v a variable, and t an expression, such that $v \notin \text{dom}(\mu)$. Then $\mu \models E[v \leftarrow t]$ iff $[v \leftarrow t] \circ \mu \models E$.*

Proof: Easy induction on the structure of terms.

If σ and τ are two substitutions, we say $\sigma \leq \tau$ iff there exists some substitution ρ such that $\tau = \sigma \circ \rho$. We say that σ is *more general* than τ.

If E is a set of equations, and σ is a substitution, we say that σ is the *most general unifier* of E iff for all models μ with $\text{dom}(m) = \text{vars}(E)$, $\mu \models E$ iff $\sigma \leq \mu$.

This asserts that σ is at least as general as any model of E. We cannot say directly that σ unifies E, as we do in the usual case of unification. Instead we require that the result of performing substitution σ on E leaves us with a *valid* set of equations: one in which any model is a solution. This leads us to the definition above.

We then have:

Theorem. *The language of type- and row-expressions has a decidable most general unification problem.*

Proof: We present an algorithm which has as its input a set E_0 of equations and has as its output either a most general unifier for E_0 or a signal that E_0 has no solutions. The algorithm has two variables, a set E of equations and a substitution σ. The algorithm will satisfy the following invariant:

INVARIANT:
$(1)(\forall\mu)(\mu \models E_0 \Rightarrow (\exists\nu)(\nu \models E \wedge \sigma \circ \nu = \mu))$
$(2)(\forall\mu)(\mu \models E \wedge \text{dom}(\mu) = \text{vars}(E) \Rightarrow \sigma \circ \mu \models E_0)$

The algorithm is given as follows:

Initialization: Set $E = E_0$ and σ to be the identity substitution on $\text{vars}(E_0)$.

Loop Step: If E is empty, then halt and return σ. Otherwise, choose an equation from E and transform E according to an algorithm to be given later.

Clearly, the initialization establishes the invariant. If E is empty and the invariant holds, then we claim that σ is a most general unifier of E_0. The first clause in the invariant guarantees that σ is at least as general as any model of E_0. To get the converse implication, assume that μ is a model such that $\sigma \leq \mu$. Then there is a substitution μ' such that $\sigma \circ \mu' = \mu$. Since μ is a model, we can choose μ' to be a model also. Now, since E is empty, $\mu' \models E$, so, by the second clause in the invariant, $\sigma \circ \mu' \models E_0$. But $\sigma \circ \mu' = \mu$, so we have $\mu \models E_0$, as desired.

It remains to give the loop step, to show that it preserves the invariant, and to show that the algorithm halts. The halting measure will be the lexicographic ordering $\langle$number of variables in E, number of symbols in $E\,\rangle$, as in the conventional unification algorithm. We now give each possibility. We begin with the cases for equations between type expressions. These are effectively the same as in the case of ordinary unification; we write them out in detail to illustrate the use of the invariant.

(1) The equation is of the form $\langle$base type$\rangle = \langle$base type$\rangle$. If the two sides are identical, delete the equation. If they are not identical, halt: there is no solution. Since there is no solution for E, by the first clause of the invariant there is no solution for E_0.

(2) The equation is of the form $t_1 \to t_2 = u_1 \to u_2$. Delete this equation from E and add the two equations $t_1 = u_1$ and $t_2 = u_2$. This is the usual step in the conventional case.

This clearly decreases the halting measure. To check the preservation of the first clause of the invariant, let E' denote the new set of equations, and let μ be a solution of E_0. By the first clause, there is a model ν such that $\sigma \circ \nu = \mu$ and $\nu \models E$. But clearly $\nu \models E'$ as well. To check that the second clause is preserved, let μ be a solution of E'. Then μ is clearly a solution to E. Hence, by the invariant, $\sigma \circ \mu \models E_0$, and the invariant is preserved.

(3) The equation is of the form $\Sigma\rho = \Sigma\rho'$ or $\Pi\rho = \Pi\rho'$. In either case, replace the equation by $\rho = \rho'$. The proof is as in the preceding case.

(4) The equation is of the form $v = t$ or $t = v$, where v is a type variable. If v appears in t, halt: there is no solution. Otherwise, replace σ by $\sigma \circ [v \leftarrow t]$, delete the equation from E, and then substitute t for all occurrences of v in the resulting set of equations.

Let E^- denote the set of equations after the selected equation has been deleted. Then the new set of equations is $E' = E^-[v \leftarrow t]$. Since v does not appear in t, it does not appear in the new set of equations. Therefore this step decreases the number of variables, and therefore decreases the halting measure. To confirm that the first clause of the invariant is preserved, let μ be a solution to E_0. Hence there is a solution ν to E such that $\sigma \circ \nu = \mu$. Since E contains $v = t$, ν must be of the form $[v \leftarrow t] \circ \nu'$, where $v \notin \mathrm{dom}(\nu')$. Hence there exists ν' such that $\mu = \sigma \circ [v \leftarrow t] \circ \nu' = \sigma' \circ \nu'$ and $[v \leftarrow t] \circ \nu' \models E^-$. Since $v \notin \mathrm{dom}(\nu')$, by the substitution lemma we have $\nu' \models E^-[v \leftarrow t]$, as desired.

To confirm that the second clause of the invariant is preserved, let μ' be a model such that $\mu' \models E'$ and $\mathrm{dom}(\mu') = \mathrm{vars}(E')$. Since v does not appear in t, we know that $v \notin \mathrm{vars}(E') = \mathrm{dom}(\mu')$. Hence we can use the substitution lemma to reason as follows:

$$\begin{aligned}
&\mu' \models E' \\
&\Rightarrow \mu' \models E^-[v \leftarrow t] \\
&\Rightarrow [v \leftarrow t] \circ \mu' \models E^- \\
&\Rightarrow [v \leftarrow t] \circ \mu' \models E \\
&\Rightarrow \sigma \circ [v \leftarrow t] \circ \mu' \models E_0 \\
&\Rightarrow \sigma' \circ \mu' \models E_0
\end{aligned}$$

as desired.

(5) Any other equation between type-expressions has no solution. Halt and report no solution.

The remaining possibility is that the chosen equation is an equation between row-expressions. In this case, the first step is to transform each expression into the form

$$t_0[a_1 \leftarrow t_1][a_2 \leftarrow t_2] \ldots [a_n \leftarrow t_n]$$

where t_0 is either a row-variable or the constant **emptyrow**, and the a_i are distinct, using the familiar identities:

$$\begin{aligned}
\rho[a \leftarrow t][a \leftarrow u] &= \rho[a \leftarrow u] \\
\rho[a \leftarrow t][b \leftarrow u] &= \rho[b \leftarrow u][a \leftarrow t] \qquad (a \neq b)
\end{aligned}$$

We now consider the possibilities, depending on whether each side of the equation begins with a variable or a constant.

(6) The equation is of the form $\rho = t$, where ρ is a row variable. If ρ appears in t, then halt and report no solution, as before, as this equation is unsatisfiable. Otherwise, proceed as in step 4 above.

(7) The equation is of the form

$$\begin{aligned}
&\mathbf{emptyrow}[a_1 \leftarrow t_1] \ldots [a_n \leftarrow t_n] \\
&\quad = \mathbf{emptyrow}[b_1 \leftarrow u_1] \ldots [b_m \leftarrow u_m]
\end{aligned}$$

This equation has a solution only if $n = m$ and each $a_i = b_i$, since the a_i and b_i completely determine the domains of the rows. If this condition is not met, halt and report no solution. Otherwise, delete this equation from E and add the equations $t_i = u_i$ for $i = 1, \ldots, n$. This step is justified in the same way as case (2) above.

(8) The equation is of the form

$$\mathbf{emptyrow}[a_1 \leftarrow t_1] \ldots [a_n \leftarrow t_n] = \rho[b_1 \leftarrow u_1] \ldots [b_m \leftarrow u_m]$$

Let A denote the set of labels $\{a_i\}$ and B denote the set of labels $\{b_i\}$. In any model of this equation, the domain of the row is exactly A, so in order to have a solution we must have $B \subseteq A$. If this condition is not met, halt and report no solution. Otherwise, without loss of generality assume that $a_1 = b_1, \ldots a_p = b_p$. Delete the current equation from E and add to E the equations $t_i = u_i$, for $1 \leq i \leq p$ and also the equation $\rho = \mathbf{emptyrow}[a_{p+1} \leftarrow t_{p+1}] \ldots [a_n \leftarrow t_n]$. This step is justified in the same way as case (2) above.

(9) The equation is of the form

$$\rho[a_1 \leftarrow t_1] \ldots [a_n \leftarrow t_n] = \rho'[b_1 \leftarrow t_1] \ldots [b_m \leftarrow t_m]$$

Again, let A denote the set of labels $\{a_i\}$ and B denote the set of labels $\{b_i\}$, and assume without loss of generality, that $a_1 = b_1, \ldots a_p = b_p$, and all labels appearing to the right of these are distinct. In any solution μ of this equation, the domain of the term must be a superset of $A \cup B$. Therefore, any solution must be of the form

$$\begin{aligned}
\rho &= \rho_0[b_{p+1} \leftarrow u_{p+1}] \ldots [b_m \leftarrow u_m] \\
\rho' &= \rho_0[a_{p+1} \leftarrow t_{p+1}] \ldots [a_n \leftarrow t_n]
\end{aligned}$$

where ρ_0 is a fresh row variable. If ρ appears on the right-hand side of the first equation or ρ' appears on the right-hand side of the second equation, then halt and report no solution. Otherwise, delete the current equation from E, add the equations $t_1 = u_1, \ldots, t_p = u_p$, and perform the substitution

$$\begin{aligned}
&[\rho \leftarrow \rho_0[b_{p+1} \leftarrow u_{p+1}] \ldots [b_m \leftarrow u_m], \\
&\quad \rho' \leftarrow \rho_0[a_{p+1} \leftarrow t_{p+1}] \ldots [a_n \leftarrow t_n]]
\end{aligned}$$

on the resulting set of equations.

Since this step is a combination of decomposition and substitution, it preserves the invariant by the same argument as steps (2) and (4) above. It decreases the halting measure because it eliminates two variables ρ and ρ' and introduces a single fresh variable ρ_0, thus decreasing the number of variables by one.

These are all the possible cases, so the algorithm is finished. *QED*.

4. Reducing Type Inference to Unification

We next turn to the reduction of type inference to unification. We next give the algorithm, in two levels of refinement. We first give its control structure and state the invariant which its inner loop must maintain. We then state the actions of the inner loop and show that they maintain this invariant. The presentation is similar to that for the unification algorithm above. Instead of analyzing a set E of equations and producing a substitution σ, the reduction algorithm analyzes a set G of subgoals which are *type-expression assertions*, and produces a set E of equations.

If M is a term, A is a set of type hypotheses whose domain is the set of free variables of M, and t is a type expression, we call the tuple (A, M, t) a *type-expression assertion*. These type expression assertions are the basic quantities manipulated by the reduction algorithm.

Typing Rules:

$$A \vdash x : \tau \quad (A(x) = \tau)$$

$$\frac{A[x \leftarrow \tau_1] \vdash M : \tau_2}{A \vdash (\lambda x.M) : \tau_1 \rightarrow \tau_2}$$

$$\frac{A \vdash M : \tau_1 \rightarrow \tau_2 \quad A \vdash N : \tau_1}{A \vdash (M\ N) : \tau_2}$$

Action Rules:

$$\langle A, x, t \rangle \Rightarrow t = A(x)$$

$$\begin{aligned} \langle A, (\lambda x.M), t \rangle \Rightarrow\ & t = \tau_1 \rightarrow \tau_2 \\ & \langle (A[x \leftarrow \tau_1])_M, M, \tau_2 \rangle \\ & (\tau_1, \tau_2 \text{ fresh}) \end{aligned}$$

$$\begin{aligned} \langle A, (M\ N), t \rangle \Rightarrow\ & \langle A_M, M, \tau_1 \rightarrow t \rangle \\ & \langle A_N, N, \tau_1 \rangle \\ & (\tau_1 \text{ fresh}) \end{aligned}$$

Figure 2. Typing and Action Rules for Simple Types

Theorem. *Given any (A_0, M_0, t_0), we can effectively construct a set E of equations such that for any set B of type assumptions and any type t,*

$$B \vdash M : t \Leftrightarrow (\exists \mu)(\mu \models E \wedge B = A_0\mu \wedge t = t_0\mu)$$

Proof: The algorithm mimics the construction of the derivation. At every step it keeps track of a set G of subgoals, which are type-expression assertions to be proved, and set E of equations between type expressions, which are verification conditions which must be satisfied for the derivation to be legal.

We need to state the invariant relating the possible completions of the derivation with the possible typings of M_0. To do this, we need to formulate the notion of a *solution* of (E, G):

Let μ be a model. If (A, M, t) is a type-expression assertion, we write $\mu \models (A, M, t)$ iff $A\mu \vdash M : t\mu$, and if G is a set of type-expression assertions, we write $\mu \models G$ iff $\mu \models (A, M, t)$ for each $(A, M, t) \in G$. Finally, we say μ *solves* (E, G) (and write $\mu \models (E, G)$) iff $\mu \models E$ and $\mu \models G$.

Thinking of μ as representing an instance of the derivation represented by (E, G), we see that $\mu \models (E, G)$ iff μ is a solution to E and μ yields a provable typing for each subgoal in G.

We can now state the invariant for our algorithm. It is just the proposition that the solutions for (E, G) generate exactly the typings of (A_0, M_0, t_0):

INVARIANT:

(1) $(\forall \mu)(\mu \models (E, G) \Rightarrow A_0\mu \vdash M_0 : t_0\mu)$

(2) $B \vdash M_0 : t \Rightarrow (\exists \mu)(\mu \models (E, G) \wedge B = A_0\mu \wedge t = t_0\mu)$

The first clause states that any solution for (E, G) generates only correct typings for (A_0, M_0, t_0) (soundness), and the second clause states that every typing of M_0 is generated by some solution for (E, G) (completeness). This is not quite an if-and-only-if, since (E, G) may involve type variables not in the initial assertion.

We may now state the top-level structure of the algorithm:

Initialization: Set $E = \emptyset$ and $G = \{(A_0, M_0, t_0)\}$. variables.

Loop Step: If $G = \emptyset$, then halt and return E. Otherwise, choose a subgoal (A, M, t) from G, delete it from G, and add to E and G new verification conditions and subgoals, as specified in an *action table* (to be supplied later).

The invariant is clearly established by our initialization step. At termination, when $G = \emptyset$, we have

(1) $(\forall \mu)(\mu \models E \Rightarrow A_0\mu \vdash M_0 : t_0\mu)$

(2) $B \vdash M_0 : t \Rightarrow (\exists \mu)(\mu \models E \wedge B = A_0\mu \wedge t = t_0\mu)$

so that the solutions of E give the typings of M_0, as desired.

We next turn to the creation of the action table. Figure 2 illustrates the typing rules and actions for the simply-typed lambda-calculus. (Here A_M denotes A restricted to the free variables of M). Since the rules always decrease the size of the terms M, they are guaranteed to terminate.

We next must show that these action rules preserve the invariant. We repeat the argument from [Wand 87]: It is easy to show that the algorithm generates only correct typings (the first clause of the invariant). We therefore only consider the

$$\langle A, M \textbf{ with } a := N, t\rangle \Rightarrow t = \Pi(\rho[a \leftarrow \tau])$$
$$\langle A_M, M, \Pi\rho\rangle$$
$$\langle A_N, N, \tau\rangle$$
$$(\rho, \tau \text{ fresh})$$

$$\langle A, \textbf{null}, t\rangle \Rightarrow t = \Pi\,\textbf{emptyrow}$$

$$\langle A, M.a, t\rangle \Rightarrow \langle A, M, \Pi(\rho[a \leftarrow t])\rangle$$
$$(\rho, \tau \text{ fresh})$$

$$\langle A, \textbf{inj}\, a\, M, t\rangle \Rightarrow t = \Sigma(\rho[a \leftarrow \tau])$$
$$\langle A, M, \tau\rangle$$
$$(\rho, \tau \text{ fresh})$$

$$\langle A, \textbf{ifcase}\, a\, M\, N\, P, t\rangle \Rightarrow \langle A, M, \Sigma\rho[a \leftarrow \tau]\rangle$$
$$\langle A, N, \tau \to t\rangle$$
$$\langle A, P, t\rangle$$
$$(\rho, \tau \text{ fresh})$$

Figure 3. Action Rules for Labelled Records and Variants

second clause of the invariant (completeness). In each case, we assume that clause (2) holds before the action is taken, and we need to show that it holds afterwards. To do this, assume that $B \vdash M_0 : t$. By the induction hypothesis, we know that

$$(\exists\mu)(\mu \models (E, G) \wedge B = A_0\mu \wedge t = t_0\mu)$$

and we need to show that

$$(\exists\mu')(\mu' \models (E', G') \wedge B = A_0\mu' \wedge t = t_0\mu')$$

where (E', G') is the state after the action step. In each case, let G_1 denote G after the selected goal has been deleted. Then we know that $\mu \models E$, $\mu \models G_1$, and $\mu \models g$, where g is the selected subgoal. We consider each case of the action table in turn.

(1) The selected subgoal is of the form (A, x, t), where x is a variable. Then $\mu \models (A, x, t)$. Hence $A\mu \vdash x : t\mu$. By the typing rules, it must be true that $A\mu(x) = t\mu$, hence $\mu \models A(x) = t$. So $\mu \models (E', G')$ as desired.

(2) The selected subgoal is of the form $(A, (M\,N), t)$. Hence $A\mu \vdash (M\,N) : t\mu$. By the typing rules, there must be some type t_1 such that $A\mu \vdash M : t_1 \to t\mu$ and $A\mu \vdash N : t_1$. So let τ_1 be a fresh type variable (not in the domain of μ), and let μ' be defined by $\mu' = \mu[\tau_1 \leftarrow t_1]$. Then $\mu' \models (A, M, \tau_1 \to t)$ and $\mu' \models (A, N, \tau_1)$, so $\mu' \models G'$, as desired.

(3) The selected subgoal is of the form $(A, (\lambda x.M), t)$. Hence $A\mu \vdash (\lambda x.M) : t\mu$. By the typing rules, there must be some types t_1 and t_2 such that $A\mu[x \leftarrow t_1] \vdash M : t_2$ and $t\mu = t_1 \to t_2$. So let τ_1 and τ_2 be fresh type variables, and let $\mu' = \mu[\tau_1 \leftarrow t_1][\tau_2 \leftarrow t_2]$. Then $\mu' \models ((A[x \leftarrow \tau_1])_M, M, \tau_2)$ and $\mu' \models t = \tau_1 \to \tau_2$, as desired.

We can now proceed to give the action rules for records and variants. These are shown in Figure 3. Here A ranges over type-expression assumptions, M, N, P range over terms, t ranges over type expressions, and ρ, τ are fresh row and type variables as they are introduced.

The reasoning for these rules is analogous to those for the simple case, comparing Figure 3 with the typing rules in Section 2. The only trick is in the rules for field selection and **ifcase**, where we have written $\rho[a \leftarrow \tau]$ to require that a be in the domain of the row in question and to have a placeholder τ for the value of a row-application ρa. Again, termination is guaranteed, since the rules always replace a term by some of its subterms. *QED.*

We can now state the solution for the type inference problem:

Theorem. *Given a term M, we can decide whether there is any A and τ such that $A \vdash M : \tau$. Furthermore, if there is, we can effectively produce a set A_0 of type hypotheses, a substitution σ and a variable $t_0 \in \mathrm{dom}\,\sigma$ such that for any set B of type assumptions and any type t,*

$$B \vdash M : t \Leftrightarrow (\exists\mu)(\sigma \leq \mu \wedge B = A_0\mu \wedge t = t_0\mu)$$

Proof: Let A_0 bind each free variable of M to a fresh type variable, and let t_0 be a fresh type variable. We then use the reduction algorithm to produce a set of equations E such that

$$B \vdash M : t \Leftrightarrow (\exists\mu)(\mu \models E \wedge B = A_0\mu \wedge t = t_0\mu)$$

We can without loss of generality assume that $\mathrm{dom}(\mu) = \mathrm{vars}(E)$. We then use the unification algorithm to produce the substitution σ such that $\mu \models E \Leftrightarrow \sigma \leq \mu$. *QED.*

Let us do a short example to illustrate this. Consider the term $\lambda u.2 \times (u.a)$. We trace the algorithm by giving a table. Each line in the table consists of two entries. The first is the current set G of subgoals. At each step we apply the action table to the first subgoal in the current goal list. This yields the goal list in the next line and some equations, which are displayed in the second entry of the next line. Interspersed are comments on the actions performed.

We will need to expand the algorithm slightly to deal with integers. To do this, we will assume the existence of a constant *twice* of type int → int. We therefore set the initial type environment A_0 to $\{(twice : (\text{int} \to \text{int}))\}$. When we begin, the goal list consists of a single goal, consisting of the type assumption A_0, the term to be typed, and a single type variable.

initial configuration
$\{((twice : (\text{int} \to \text{int})), \lambda u.twice\,(u.a), t_0)\}$
analyzing λ
$\{((u : t_1,\ twice : (\text{int} \to \text{int})), twice\,(u.a), t_2)\};\quad t_0 = t_1 \to t_2$
splitting combination
$\{((twice : (\text{int} \to \text{int})), twice, t_3 \to t_2), ((u : t_1), u.a, t_3)\}$
deleting goal for *twice*
$\{((u : t_1), u.a, t_3)\};\quad t_3 \to t_2 = \text{int} \to \text{int}$
analyzing selection
$\{((u : t_1), u, \Pi(\rho[a \leftarrow t_3]))\}$
deleting goal for u
$\emptyset;\quad \Pi(\rho[a \leftarrow t_3]) = t_1$

Thus the equations generated are:

$$t_0 = t_1 \to t_2$$
$$t_3 \to t_2 = \text{int} \to \text{int}$$
$$\Pi(\rho[a \leftarrow t_3]) = t_1$$

Solving these equations using the algorithm of the previous section yields

$$\Pi(\rho[a \leftarrow \text{int}]) \to \text{int}$$

as the principal type of the term, as desired.

5. Polymorphic Definitions

The system as described so far is entirely *monomorphic*: though terms may have more than one type, the only binding mechanism is lambda-binding, which, just as in the simply-typed lambda-calculus, allows only monomorphic values to be passed. Thus there is no way to take advantage of the polymorphism in the notation.

This capability is provided in ML through the **let** mechanism. For the purposes of this paper, we will present a simplified version of this mechanism which will work only at the top level. Define the set of *programs* as follows:

$$\langle\text{program}\rangle ::= \langle\text{definition}\rangle; \langle\text{program}\rangle \mid \langle\text{term}\rangle$$
$$\langle\text{definition}\rangle ::= \langle\text{identifier}\rangle\ =\ \langle\text{term}\rangle$$

For type-checking purposes, a definition associates the identifier with the principal type of the term on the right-hand side. The scope of the definition is the ⟨program⟩ that follows it. In this scope, the identifier is polymorphic, and may be used at any of the instances of its principal type. Consider, for example, the program:

$$twicea = \lambda u.2 \times (u.a);$$
$$twicea[a := 3, b := \text{true}] + twicea[a := 4, b := 7]$$

where $[a := M, b := N]$ abbreviates **null with** $a := M$ **with** $b := N$. Here *twicea* gets assigned principal type $\Pi(\rho[a \leftarrow \text{int}]) \to \text{int}$, and is used with two different instances of ρ: with $\rho = \Pi((a, \text{int}), (b, \text{bool}))$ and with $\rho = \Pi((a, \text{int}), (b, \text{int}))$.

This is how polymorphism allows us to deal with inheritance: *twicea* may be applied to any object (that is, any record) containing an a component which is an integer. Thus *twicea* is applicable to any descendant of $\Pi((a, \text{int}))$.

One might wish to rewrite the typing rule for definitions as

$$\frac{A \vdash M[N/x]: \tau}{A \vdash (x = N; M): \tau}$$

While this rule gives the same typings in this system, it is not equivalent in general. The issue is whether the type system provides enough types to abstract the possible uses of x; re-expanding each occurrence of x, as if it were a macro, prevents any sort of separate compilation of M and prevents us from addressing this issue at all. In the next section, we shall see how this makes a difference.

This mechanism for introducing polymorphism has been used successfully by ML [Gordon *et al.* 78] and by SPS [Wand 84]. In practice, it seems quite adequate for ordinary programming. More complicated situations, such as separate compilation of polymorphic modules, require full-fledged quantified types, for which type inference is a long-standing open problem.

6. Comparison with Cardelli's System

The system C84 uses the same types as ours. It does not use type expressions to express polymorphism, but instead adds a subtype relation between types, which is defined as follows:

$$\rho_1 < \rho_2 \iff \text{dom}\,\rho_1 \subset \text{dom}\,\rho_2 \wedge (\forall a \in \text{dom}\,\rho_1)[\rho_1 a < \rho_2 a]$$
$$\tau_1 \to \tau_2 < \tau_1' \to \tau_2' \iff \tau_1' < \tau_1 \wedge \tau_2 < \tau_2'$$
$$\Sigma\rho < \Sigma\rho' \iff \rho < \rho'$$
$$\Pi\rho < \Pi\rho' \iff \rho' < \rho$$

C84 uses explicit types on all lambda variables ($\lambda x{:}\tau.M$ instead of $\lambda x.M$). Polymorphism is obtained by altering the rule for typing of individual variables to read:

$$A \vdash x : \tau' \quad (A(x) = \tau, \tau < \tau')$$

To see how this deals with inheritance, let us consider again the program we wrote before:

$$twicea = \lambda u.2 \times (u.a);$$
$$twicea[a := 3, b := \text{true}] + twicea[a := 4, b := 7]$$

This program is well-typed; we assign *twicea* the type

$$\Pi((a, \text{int})) \to \text{int}$$

and we use it at the two types $\Pi((a, \text{int}), (b, \text{bool})) \to \text{int}$ and $\Pi((a, \text{int}), (b, \text{int})) \to \text{int}$, each of which is greater than the original type.

One can still ask about type inference, however: when is an untyped term M the image under type-erasing of a typed term? One can then begin to compare the properties of the two systems.

The original version of C84 included some things which are not relevant to our current concerns (such as recursive type definitions), and had a somewhat different treatment of record construction and case selection (as noted in Section 2). In order to be precise, for the remainder of this section, we will define C84 to be our system with the rule for typing identifiers modified as shown. This will allow us a moderately fair comparison between the systems and their treatment of polymorphism.

For the pure systems (without definitions), the terms typed by our system are a subset of those typed by C84. To show the inclusion is proper, simply consider

$$((\lambda f.f([a := 3, b := \text{true}]) + f([a := 4, b := 7]))\ \ twicea)$$

which is well typed in C84 (assign f the type $\Pi((a, \text{int})) \to \text{int}$), but is not well-typed in the current system, since the two instances of f require different values of ρ in $\Pi(\rho[a \leftarrow \text{int}]) \to \text{int}$.

We next turn to a comparison of the systems with polymorphic definitions. Since C84 provides no type expressions, we choose to interpret definitions as simple lambda-bindings, as follows:

$$\frac{A \vdash M : \tau \quad A[x \leftarrow \tau] \vdash N : \tau'}{A \vdash (x = M; N) : \tau'}$$

The polymorphism is provided by the standard identifier-lookup mechanism.

This is not equivalent to using the "macro-expansion" rule for definitions. The interesting issue is precisely whether there is a type τ which allows for all the uses of x in N; macro-expanding each occurrence of x avoids the issue completely.

For the systems with definitions, there are programs which are well-typed in the the current system but which have no type in C84. Consider, for example

$$\begin{aligned} &\mathit{doublea} = \lambda u.u \textbf{ with } a := 2 \times (u.a); \\ &(\mathit{doublea}[a := 1; b := 2]).b + (\mathit{doublea}[a := 3; c := 4]).c \end{aligned}$$

This is well-typed in the current system, since *doublea* has principal type

$$\Pi(\rho[a \leftarrow \text{int}]) \to \Pi(\rho[a \leftarrow \text{int}])$$

but it is not well-typed in C84. To see this, consider the types of the two occurrences of *doublea*:

$$\Pi((a, \text{int}), (b, \text{int})) \to \Pi((a, \text{int}), (b, \text{int}))$$

and

$$\Pi((a, \text{int}), (c, \text{int})) \to \Pi((a, \text{int}), (c, \text{int}))$$

The greatest lower bound of these two types is

$$\Pi((a, \text{int})) \to \Pi((a, \text{int}), (b, \text{int}), (c, \text{int}))$$

which is not a possible type for *doublea*. Hence this program has no type in C84.

Hence the systems with polymorphic definition are incomparable.

We can sharpen these observations somewhat by considering the pure expressiveness of the two kinds of polymorphism. Let us say a set S of types is definable in our system iff there exists a type t such that $u \in S$ iff u is a substitution instance of t. Say a set S of types is definable in C84 iff there exists a type t such that $u \in S$ iff $t < u$. Then the definitional power of the two systems is incomparable. The previous example shows that there is a set of types which is definable in the current system but not in C84. Similarly, the set of types defined in C84 by

$$\Pi((a, \text{int}), (b, \text{int})) \to \Pi((a, \text{int}), (b, \text{int}))$$

is not definable in the current system, since it has no way to specify a class of rows containing *fewer* elements than $((a, \text{int}), (b, \text{int}))$ using only a single principal type.

It is straightforward to add inclusions to the algorithm which generates the verification conditions, but it is no longer obvious that the resulting set of equalities and inclusions has a decidable satisfiability problem.

The system we have proposed seems adequate for use in programming systems relying on "structural" inheritance through shared field names. On the other hand, it is by no means clear that this is an appropriate model of inheritance for other object-oriented systems. For example, in our introductory example, there is no guarantee that the `id-number` fields for submarines and weapons systems will be compatible. Hence a function that manipulates the `id-number` field might not really be applicable to both submarines and weapons-systems. Many real object-oriented programming systems use *name* inheritance, in which the inheritance can be more tightly controlled. This leads to a somewhat different theory, which we hope to present elsewhere.

References

[Cardelli 84]
Cardelli, L. "A Semantics of Multiple Inheritance," *Semantics of Data Types, International Symposium*, Springer Lecture Notes in Computer Science *173* (1984), 51–68.

[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.

[Gordon, *et al.* 78]
Gordon, M., Milner, R., Morris, L., Newey, M., and Wadsworth, C. "A Metalanguage for Interactive Proof in LCF," *Proc. 5th Annual ACM Symp. on Principles of Programming Languages* (1978) 119–130.

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

[Wand 84]
Wand, M. "A Semantic Prototyping System," *Proc. ACM SIGPLAN '84 Compiler Construction Conference* (1984) 213–221.

[Wand 86]
Wand, M. "A Simple Algorithm and Proof for Type Inference," *Fundamenta Informaticae 10* (1987), to appear.