A Simple Algorithm and Proof for Type Inference

Mitchell Wand*

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

This paper appeared in *Fundamenta Informaticae 10* (1987), 115-122, and should be cited as such.

Abstract

We present a simple proof of Hindley's Theorem: that it is decidable whether a term of the untyped lambda calculus is the image under type-erasing of a term of the simply typed lambda calculus. The proof proceeds by a direct reduction to the unification problem for simple terms. This arrangement of the proof allows for easy extensibility to other type inference problems.

1. Introduction

The type inference problem may be stated simply: Given a term of the untyped lambda calculus, to find all terms of the typed lambda calculus which yield the given term when the type information on bound variables is deleted. Since such terms can differ only in their types, this problem is sometimes referred to as finding the "possible typings" of a term.

This problem was first formulated and solved by Curry (probably in the 1930's) [Curry & Feys 58, Sec. 8C] and Hindley [Hindley 69]. The problem was independently reformulated by Morris [Morris 68] and Milner [Milner 78] in its computer science context: the reconstruction of type information by a compiler. See [Hindley 83] for an insightful discussion of the various motivations for the problem.

Milner was the first to make the connection with the unification problem [Robinson 65 (though the algorithm goes back to J. Herbrand [Robinson 86])]. The proofs of Curry and Hindley, in particular, involve a tedious reconstruction of the unification mechanism (Hindley, for example, refers to "highest common instances."). As a result, the proofs are long and difficult. Milner's original paper [Milner 78] does not prove the correctness of his algorithm, though he does state a semantic soundness result. The correctness of the algorithm is claimed without proof in [Damas & Milner 82]; the proofs are relegated

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

to Damas's unpublished thesis. [Reynolds 85] sketches the first short proof. Cardelli [Cardelli 85] gives an excellent discussion of the inference problem and shows how it can be reduced to the unification problem, but presents no proofs.

In this paper, we make the reduction precise by formally presenting an algorithm for the reduction. We prove its correctness by stating an invariant relating the intermediate states of the computation to the desired result, and then proving that our algorithm preserves this invariant.

The algorithm proceeds in the manner of a verification-condition generator. It attempts to recreate the derivation tree for the term, emitting along the way some verification conditions (in this case, equations). The invariant of the algorithm ensures that at every stage we are generating the "most general" such derivation tree.

By stating this algorithm and invariant in a way which is both formal and independent of the details of the typing rules, we build a base on which extensions of the method can be made.

Preliminaries are given in Section 2. In Section 3, we state the skeleton of the algorithm, and compare it with Milner's algorithm and with Reynolds'. In Section 4, we apply our skeleton to the simply-typed lambda calculus. Section 5 contains some extensions and concluding remarks.

2. Preliminaries

We assume we have a set V of variable symbols (typically x, y, z, etc.).

Definition. The set Λ of untyped lambda terms is defined inductively by:

$$(1)\ V \subset \Lambda$$
$$(2)\ M, N \in \Lambda \Rightarrow (M\ N) \in \Lambda$$
$$(3)\ M \in \Lambda, x \in V \Rightarrow (\lambda x.M) \in \Lambda$$

(We assume the presence of a suitable limiting clause in all inductive definitions).

We assume we have a set O of symbols (called *basic types*), and a set TV of *type variables* (typically $\tau_1, \tau_2, \ldots$).

Definition. The set T of *types* is defined inductively by

$$(1)\ O \subset T$$
$$(2)\ t_1, t_2 \in T \Rightarrow (t_1 \rightarrow t_2) \in T$$

Definition. The set Λ_T of *typed lambda-terms* is obtained by modifying the modifying clause (3) in the definition of Λ to read: $M \in \Lambda_T, x \in V, t \in T \Rightarrow (\lambda x : t.M) \in \Lambda_T$.

The set of free variables of a term in Λ or Λ_T is defined in the usual way. Note that the elements of Λ_T need not be "well-typed" in the usual way. A *type assertion* is intended to express the proposition that a term is well-typed:

Definition. A *type assumption* is a partial function $A : V \to T$ with finite domain.

Definition. A *type assertion* is a triple (A, M, t), where A is a type assumption, M is a lambda-term (either typed or untyped, depending on context), and t is a type, and where the domain of A is exactly the free variables of M.

The last condition in this definition is a technical one which makes the proofs slightly neater.

We write A_M for A restricted to the free variables of M. We write $A[x \leftarrow t]$ for the extension of the function A in which the variable x is associated with the type t.

We next give the typing rules for the typed lambda calculus. If the assertion (A, M, t) is provable in these typing rules, we write $A \vdash M : t$ and say that M has type t under assumptions A. As we shall see, the type assumptions determine the types of the free variables in M.

The typing rules follow the definition of Λ_T:

$$A \vdash x : t \qquad \text{where } A(x) = t$$
$$A_M \vdash M : t_1 \to t_2,\ A_N \vdash N : t_1 \Rightarrow A \vdash (M\ N) : t_2$$
$$(A[x \leftarrow t_1])_M \vdash M : t_2 \Rightarrow A \vdash (\lambda x : t_1.M) : t_1 \to t_2$$

The type assumption in the hypothesis of the last rule is designed to ensure that it involves a legal type assertion.

For type inference, we are concerned with the corresponding deduction system for Λ, in which we deal with type assertions for untyped terms, in which the type information has been erased:

$$A \vdash x : t \qquad \text{where } A(x) = t$$
$$A_M \vdash M : t_1 \to t_2,\ A_N \vdash N : t_1 \Rightarrow A \vdash (M\ N) : t_2$$
$$(A[x \leftarrow t_1])_M \vdash M : t_2 \Rightarrow A \vdash (\lambda x.M) : t_1 \to t_2$$

The type inference problem (for closed terms) may then be stated as follows: Given a closed term M in Λ, find all types t such that $\emptyset \vdash M : t$. We can characterize these types using the mechanism of *type expressions*:

Definition. The set TE of *type expressions* is defined inductively by adding to the definition of T the clause $TV \subset TE$.

We will also allow type expressions to appear in type assertions; we call the resulting objects *type-expression assertions.*

We define substitutions σ from type expressions to types in the usual way, and write $t\sigma$ for the action of a substitution on a type expression, and $A\sigma$ for its action on the type assumption A, given by $A\sigma(x) = (A(x))\sigma$.

We may now state the main theorem:

Theorem. *(1) Given a closed term M in Λ, it is decidable whether there exists a type t such that $\emptyset \vdash M : t$. (2) If there is any such t, then there is a type expression u such that the typings of M are precisely the types of the form $u\sigma$ for all substitutions σ.*

Such a u is called a *principal type* of M. It is clearly unique up to isomorphism. We will, of course, solve the problem for terms with free variables as well.

3. The Algorithm Skeleton

The central problem in any algorithm for type inference is to keep track of all possible derivations. This is done by simulating a single derivation, but using type expressions instead of types to keep track of the possible types that might occur at each node of the derivation tree.

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 may now give the structure of the algorithm:

Algorithm (Skeleton)

Input: A term M_0 of Λ.

Initialization: Set $E = \emptyset$ and $G = \{(A_0, M_0, t_0)\}$, where t_0 is a type variable and A_0 maps the free variables of M_0 to other distinct type 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).

End of Skeleton.

We refer to this as a skeleton for an algorithm because we may complete it in different ways by using different tables of actions for processing the subgoals in the loop step.

Regardless of the choice of actions, we need to state an 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):

Definition. Let σ be a substitution. We say σ *solves* an equation e (and write $\sigma \models e$) if σ unifies e. If E is a set of equations, we write $\sigma \models E$ iff $s \models e$ for each $e \in E$. If (A, M, t) is a type-expression assertion, we write $\sigma \models (A, M, t)$ iff $A\sigma \vdash M : t\sigma$, and if G is a set of type-expression assertions, we write $\sigma \models G$ iff $\sigma \models g$ for each $g \in G$. Finally, we say σ *solves* (E, G) (and write $\sigma \models (E, G)$) iff $\sigma \models E$ and $\sigma \models G$.

Thinking of σ as representing an instance of the derivation represented by (E, G), we see that $\sigma \models (E, G)$ iff σ is a solution to E and σ 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):

$$
\begin{array}{ll}
\textit{INVARIANT}: & \\
(1) & (\forall\sigma)(\sigma \models (E, G) \Rightarrow A_0\sigma \vdash M_0 : t_0\sigma) \\
(2) & B \vdash M_0 : t \Rightarrow (\exists\sigma)(\sigma \models (E, G) \wedge B = A_0\sigma \wedge t = t_0\sigma)
\end{array}
$$

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.

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

$$
\begin{array}{ll}
(1) & (\forall\sigma)(\sigma \models E \Rightarrow A_0\sigma \vdash M_0 : t_0\sigma) \\
(2) & B \vdash M_0 : t \Rightarrow (\exists\sigma)(\sigma \models E \wedge B = A_0\sigma \wedge t = t_0\sigma)
\end{array}
$$

so that the solutions of E give the typings of M_0. Since equations between type expressions have effectively findable most general unifiers, this yields decidable principal types.

Assuming that we can complete the action table for the algorithm, we can now prove the main theorem.

Proof of Main Theorem: Use the algorithm on M. Use the unification algorithm on the resulting set of equations E. If E has no solutions, then M is not typable. Otherwise, let σ be the most general unifier of E. Then $t_0\sigma$ is the principal type for M.

It is interesting to compare this algorithm with Milner's and with Reynolds'. Milner [Milner 78] gives two algorithms, in which the production of equations is interleaved with their solution. Reynolds' algorithm [Reynolds 85] is quite different. It proceeds by induction on the syntactic structure

of lambda-terms, from the leaves (variables) toward the whole term. This algorithm involves performing unification on pairs of type assumptions, which might be quite large in a realistic system.

Note that we have proven the correctness of our algorithm *skeleton*. We can now prove correctness for any set of typing rules, so long as we can produce an action table which always terminates and preserves the invariant. In the next section, we do this for the case of the simply-typed lambda calculus.

4. Action Table and Proof for the Typed Lambda Calculus

For the typed lambda calculus, with typing rules given in Section 2, there are three kinds of actions, corresponding to the three kinds of lambda-term that might appear in the selected subgoal:

Case 1. (A, x, t). Generate the equation $t = A(x)$.

Case 2. $(A, (M\ N), t)$. Let τ_1 be a fresh type variable that appears nowhere else in (E, G). Then generate the subgoals $(A_M, M, \tau_1 \to t)$ and (A_N, N, τ_1).

Case 3. $(A, (\lambda x.M), t)$. Let τ_1 and τ_2 be fresh type variables. Generate the equation $t = \tau_1 \to \tau_2$ and the subgoal $((A[x \leftarrow \tau_1])_M, M, \tau_2)$.

Proposition. *With this action table, the algorithm always terminates.*

Proof: Each action generates subgoals involving terms smaller than the original.

Proposition. *Each action in this action table preserves the invariant of the algorithm.*

Proof: It is easy to show that the algorithm generates only correct typings (the first clause of the invariant). We therefore only consider the 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

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

and we need to show that

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

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 $\sigma \models E$, $\sigma \models G_1$, and $\sigma \models g$, where g is the selected subgoal. We consider each case of the action table in turn.

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

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

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

5. Example

To show how the algorithm behaves, let us consider typing the S combinator, $\lambda x.\lambda y.\lambda z.xz(yz)$. We trace the algorithm by giving a table. Each line in the table consists of three 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; the last entry is a comment describing the action.

When we begin, the goal list consists of a single goal, consisting of an empty type assumption, the term to be typed, and a single type variable.

$\{(\emptyset, \lambda x.\lambda y.\lambda z.xz(yz), t_0)\}$	initial configuration
$\{((x : t_1), \lambda y.\lambda z.xz(yz), t_2)\}$; $\quad t_0 = t_1 \to t_2$	analyzing λ
$\{((x : t_1, y : t_3), \lambda z.xz(yz), t_4)\}$; $\quad t_2 = t_3 \to t_4$	analyzing λ
$\{((x : t_1, y : t_3, z : t_5), xz(yz), t_6)\}$; $\quad t_4 = t_5 \to t_6$	analyzing λ
$\{((x : t_1, z : t_5), xz, t_7 \to t_6), ((y : t_3, z : t_5), yz, t_7)\}$	splitting combination
$\{((x : t_1), x, t_8 \to (t_7 \to t_6)), ((z : t_5), z, t_8), ((y : t_3, z : t_5), yz, t_7)\}$	splitting combination
$\{((z : t_5), z, t_8), ((y : t_3, z : t_5), yz, t_7)\}$; $\quad t_1 = t_8 \to t_7 \to t_6$	deleting goal for x
$\{((y : t_3, z : t_5), yz, t_7)\}$; $\quad t_8 = t_5$	deleting goal for z
$\{((y : t_3), y, t_9 \to t_7), ((z : t_5), z, t_9)\}$	splitting combination
$\{((z : t_5), z, t_9)\}$; $\quad t_9 \to t_7 = t_3$	deleting goal for y
$\emptyset$; $\quad t_9 = t_5$	deleting goal for z

Thus the equations generated are:

$$
\begin{aligned}
t_0 &= t_1 \to t_2 \\
t_2 &= t_3 \to t_4 \\
t_4 &= t_5 \to t_6 \\
t_1 &= t_8 \to t_7 \to t_6 \\
t_8 &= t_5 \\
t_9 \to t_7 &= t_3 \\
t_9 &= t_5
\end{aligned}
$$

Solving these equations by the unification algorithm gives the solution

$$t_0 = (t_5 \to t_7 \to t_6) \to (t_5 \to t_7) \to (t_5 \to t_6)$$

which is the principal type of this term.

Of course, the trimming of the type assumptions to match the free variables of the terms in the goals is not necessary in a real implementation; it is included only to facilitate the technical development.

6. Extensions and Conclusions

By formulating the type inference problem as a reduction to a unification problem, we can alter the parameters of the proof to yield a variety of extensions.

1. In this paper we have chosen our set E of verification conditions to consist of equations between finite terms over base types, type variables, and the binary constructor $\to$. Clearly the main theorem holds also for any set of types which includes these and which still has effective most general unifiers, as claimed in [Wand 84]. Examples of such type systems include: arbitrary type constructors, infinite regular types, etc.
2. The result is easily extended to type systems with products, disjoint unions, etc. Finite products and unions are straightforward; with slightly more effort, one can design a type system with principal types for systems of labelled products and unions. This in turn allows a treatment of certain kinds of object-oriented programming systems, complementary to the treatment in [Cardelli 84].
3. The invariant can also be extended to allow symbolic type analysis (analogous to symbolic execution) of entire classes of programs. This allows us to infer derived type rules for syntactic extensions of languages. The resulting analysis also allows a treatment of **let**-polymorphism in ML [Milner 78]. We hope to report on this 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," unpublished manuscript, 1985.

[Curry & Feys 58]
Curry, H.B. and Feys, R. *Combinatory Logic*, Vol. 1. North-Holland, Amsterdam, 1958.

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

[Hindley 83]
Hindley, R. "The Completeness Theorem for Typing λ-Terms" *Theoret. Comp. Sci. 22* (1983) 1–17.

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

[Damas & Milner 82]
Damas, L., and Milner, R. "Principal Type-Schemes for Functional Programs," *Conf. Rec. Ninth Ann. ACM Symp. on Principles of Programming Languages* (1982) 207–212.

[Morris, J.H. 68]
Morris, J.H. "Lambda Calculus Models of Programming Languages", MIT Ph.D. dissertation.

[Reynolds 85]
Reynolds, J.C. "Three Approaches to Type Structure," *Proc. TAPSOFT Advanced Seminar on the Role of Semantics in Software Development* (Berlin, March, 1985), Springer Lecture Notes in Computer Science.

[Robinson 65]
Robinson, J.A. "A Machine-Oriented Logic Based on the Resolution Principle," *J. Assoc. Comput. Mach. 12* (1965), 23–41.

[Robinson 86]
Robinson, J.A., presentation at Symposium on Logic in Computer Science, Cambridge, MA, June, 1986.

[Wand 84]

Wand, M. "A Types-as-Sets Semantics for Milner-style Polymorphism," *Conf. Rec. 11th ACM Symp. on Principles of Programming Languages* (1984), 158–164.