PARTIAL POLYMORPHIC TYPE INFERENCE IS UNDECIDABLE

Hans-J. Boehm

Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

Polymorphic type systems combine the reliability and efficiency of static type-checking with the flexibility of dynamic type checking. Unfortunately, such languages tend to be unwieldy unless they accommodate omission of much of the information necessary to perform type checking. The automatic inference of omitted type information has emerged as one of the fundamental new implementation problems of these languages. We show here that a natural formalization of the problem is undecidable. The proof is directly applicable to some practical situations, and provides a partial explanation of the difficulties encountered in other cases.

Introduction

It is generally accepted that compile-time type checking of programs speeds up the debugging process, improves the reliability of programs, and makes it easier for a compiler to generate good machine code. Unfortunately many early type checking systems were so rigid that they made it impossible to construct, for example, general purpose data structures. In Pascal, it is not possible to provide a general binary search tree implementation without completely defeating type-checking facilities. Separate implementations are needed for binary search trees with integer data, binary search trees with string data, etc.

Some languages, notably Ada*, address the problem by allowing the programmer to define templates for such types, which are then expanded at compile time. It is usually required that all such expansions be explicitly indicated by the user. This solves the most apparent problems, but introduces more subtle ones. It is possible to construct a "polymorphic" or "generic" implementation of binary search trees that will operate on trees with data of arbitrary, but fixed, type. It is not possible to build binary search trees with leaf data type dependent on input to the program at run-time. In the Ada scheme, it is also impossible to write another polymorphic program which will work with any general table implementation, and then to instantiate it with the (generic) binary search tree implementation. Since this introduces new notions of abstraction and instantiation, distinct from normal functional abstraction and application, it is also likely to significantly increase the size of the language. (More than half a page of the Ada grammar is concerned with "generics". See, for example, [Wie 83].)

More ambitious solutions involve directly extending the typing system to allow the assignment of a type to, say, a look up function on general binary search trees. (In the Ada case only a specific instance has a type per se.) A number of such polymorphic type checking systems have been introduced (cf. [Rey 74], [Mil 78], [Gor 79], [Boe 85a], [Mee 83], [MQu 84], [Jen 84]).

Unfortunately languages based on such systems tend to aggravate one problem only hinted at by languages such as Pascal; the full syntax includes a substantial amount of type related information, much of which is useful only to the compiler. This becomes especially serious in languages such as Russell that take a very pure approach, usually without much regard for syntactic brevity [Boe 85b].

Implementations of such languages typically allow the programmer to omit as much type related information as possible. The compiler is then required to infer this information before it can type check the program. Some relatively simple typing disciplines, such as that of [Mil 78] allow the programmer to omit all or part of such information, with the guarantee that the compiler will be able to infer it whenever possible. Such disciplines unfortunately retain at least one of the deficiencies mentioned above; they do not allow abstraction with respect to another object of polymorphic type (such as the general search tree implementation).

We show here that for some more general systems, it is impossible for the compiler to guarantee successful inference of omitted type information. That is, it becomes undecidable whether a program with some omitted type information can be completed to form a completely specified type-correct program. This result applies directly to the typing discipline of Russell (cf. [Boe 80], [Boe 85a]). It gives insight into the difficulty of the type inference problem for other disciplines, such as that of [McQ 84].

*Ada is a trademark of the U.S. Department of Defense (Ada Joint Program Office).

0272-5428/85/0000/0339$01.00 © 1985 IEEE

Background

Most recently proposed polymorphic typing systems fall into one of two categories.

Type quantification systems allow types which include explicit quantifiers. Thus the type of the identity function

$$\lambda x \,.\, x$$

would be specified as

$$\forall\, t \,.\, t \rightarrow t$$

indicating that it can be applied to an argument of any type and will return a result of that same type. The system of [Mil 78] is a special case of this scheme, in which embedded quantifiers have been disallowed.

Type abstraction systems introduce types as explicit parameters to functions. The identity function would have another parameter, namely the type of its "real" parameter. We adhere to the notation of [Rey 74], and indicate such an abstraction with respect to type by a capital Λ. (It is no longer essential to distinguish the two notions of abstraction. Indeed, more practical languages based on this system generally do not. We maintain the distinction for the sake of clarity and tradition.)

The identity function itself would be written as

$$\Lambda t \,.\, \lambda x{:}t \,.\, x$$

It must be applied to both a type and an element of that type. It returns its second argument. The type of the identity function would be

$$\Delta t \,.\, t \rightarrow t$$

(This can be read as: If the expression is applied to a type t, then the type of the resulting expression is $t \rightarrow t$.)

We will deal directly with a form of type abstraction. Our result can be carried over to the framework of [Boe 80] and [Boe 85a].

It will follow from [Lei 83]* that our result also has implications for type quantification. The formal statement of the theorem in that framework is not very appealing. On the other hand, it becomes clear that any type inference algorithm dealing with either type system must rely on the fact that certain information is *not* specified by the programmer. This disqualifies any approach resembling that of [Mil 78].

We only consider the situation in which partial typing information may be supplied by the programmer. At present it appears conceivable that the type inference problem with no programmer supplied information is easier.** This is not as surprising as it might appear at first, since the absence of information leaves the inference algorithm with more freedom in assigned types to expressions.

*We refer to theorem 4.1 of [Lei 83].

**[Lei 83] claims that the problem with no information, and without recursion, is decidable. [McC 84] gives an algorithm to solve a related problem.

Problem Statement

We will show that the type inference problem is undecidable for a simple extension of the Girard-Reynolds (cf. [Rey 74]) polymorphically typed λ-calculus. Our problem statement is based on the syntactic inference problem in [Mit 84].

The set of legal type expressions is given by the grammar:

```
<type> ::= <identifier>
         | <type> → <type>
         | Δ <identifier> . <type>
```

The function space constructor, "$\rightarrow$", associates to the right. Type expressions differing only in the names of Δ-bound variables are considered to be equivalent.

If f is a function then we define $\{y \leftarrow a\}f$ to be the function g such that

$$\begin{aligned} g(x) &= f(x) \qquad \text{if } x \neq y \\ g(y) &= a \end{aligned}$$

A type assignment is a function from a finite set of variables to type expressions. Such functions are used to specify the types of free variables in a λ-expression.

For our purposes, (fully typed) λ-expressions are defined by the following grammar.

```
<expr> ::= <identifier>
         | <function abstraction>
         | <function application>
         | <type abstraction>
         | <type application>
         | <recursion>
<function abstraction> ::=
         | λ <identifier> : <type> . <expr>
<function application> ::=
         | <expr> <expr>
<type abstraction> ::=
         | Λ <type> . <expr>
<type application> ::=
         | <expr> <type>
<recursion> ::=
         | fix <identifier> : <type> . <expr>
```

Multiple applications associate to the left. Other ambiguities will be resolved with parentheses.

The **fix** construct is taken from [Mil 78]. Informally it denotes the value of the expression, with the identifier bound to the value of the expression.

For each type assignment A we define a partial function T_A which maps λ-expressions to their types. We define T_A recursively:

1) $T_A(x)$, where x is an identifier, is A(x). (It is undefined if A(x) is not defined.)

2) $T_A(\lambda x{:}\, \tau \,.\, e)$ is $\tau \rightarrow \sigma$, where σ is $T_{\{x \leftarrow \tau\}A}(e)$.

3) If $T_A(e_1) = \tau \rightarrow \sigma$ and if $T_A(e_2) = \tau$, then $T_A(e_1\ e_2)$ is σ.

4) $T_A(\Lambda t\ .\ e)$ is $\Delta t\ .\ T_A(e)$, provided t does not occur free in any A(x).

5) If $T_A(e)$ is $\Delta t\ .\ \tau$, then $T_A(e\ \sigma)$, where σ is a type expression, is $\tau[t \leftarrow \sigma]$, that is, τ with σ substituted for t (with the usual provisions to prevent capture).

6) $T_A(\textbf{fix}\ x{:}\ \tau\ .\ e)$ is $T_{\{x \leftarrow \tau\}A}(e)$.

An expression e is type correct with respect to a type assignment A if $T_A(e)$ is defined.

A partially typed expression is one which can be obtained from a fully typed expression e by some number of applications of the following three rewrite rules to subexpressions of e:

$$\begin{array}{lcl} \lambda x{:}\ \tau\ .\ e & \rightarrow & \lambda x\ .\ e \\ \textbf{fix}\ x{:}\ \tau\ .\ e & \rightarrow & \textbf{fix}\ x\ .\ e \\ e\ \tau & \rightarrow & e\ ? \end{array}$$

If a partially typed expression e' can be obtained from the fully typed expression e, we say that e' is a *partial erasure* of e, written as *pe*(e',e). A partially typed e' is type correct with respect to an assignment A iff there is a fully typed e, which is type correct with respect to A, such that *pe*(e',e). Note that there may be many such e; thus a partially typed expression generally does not have a unique type.

We define an initial type assignment I as follows:

$$\begin{array}{lcl} I(c) & = & C \\ I(eq) & = & \Delta t\ .\ t \rightarrow t \rightarrow t \\ I(x) & & \text{is undefined otherwise} \end{array}$$

We show that it is undecidable whether a given partially typed expression is type correct with respect to I.

We have deviated from standard practice in not allowing arguments to type applications to be completely deleted, but only to be replaced by the place-holder "?". Unfortunately, the following proof relies on this fact. It is easily argued that this is an appropriate model of many practical type inference problems. In languages allowing functions with more than one parameter, we are commonly interested in inferring a missing type argument, in the presence of other non-type arguments. It is difficult to conceive of a situation in such a language where it is important to have the system infer the existence of a type application. Furthermore, it could be argued that an inference algorithm which does not allow the user to specify the placement of type applications, does not necessarily enforce those typing constraints intended by the user.

An Example

The following example is intended to give some motivation for the proof of the next section. It should also give some intuition for the fact that this problem cannot be solved with a unification based approach, as in [Mil 78].

To facilitate the discussion in the remainder of the paper, we introduce two abbreviations.

We use "proj" to denote the partially typed λ-expression

$$(\Lambda\sigma\ .\ \Lambda\tau\ .\ \lambda x{:}\sigma\ .\ \lambda y{:}\tau\ .\ y)\ ?\ ?$$

The partially typed λ-expression

$$\text{proj}\ e_1\ e_2$$

is type correct if and only if both e_1 and e_2 are type correct. Furthermore, if e_2 is completely typed, then the type of the composite expression must be the type of e_2.

The block

$$\begin{array}{l} \textbf{let} \\ \quad x_1 = e_1 \\ \quad x_2 = e_2 \\ \quad \ldots \\ \quad x_n = e_n \\ \textbf{in} \\ \quad e \end{array}$$

abbreviates

$$(\lambda x_1\ .\ \lambda x_2\ .\ \ldots\ \lambda x_n\ .\ e)\ e_1\ e_2\ \ldots\ e_n$$

Note that if e and the e_i are completely typed expressions, then the type of the **let**-block with respect to A is

$$T_{\{x_1 \leftarrow T_A(e_1)\}\ \ldots\ \{x_n \leftarrow T_A(e_n)\}\ A}(e)$$

since the application is type correct only if the missing parameter types are the types of the e_i.

Let

$$\lambda^1 y{:}t\ .\ e$$

denote

$$\lambda y{:}t\ .\ \lambda y{:}t\ .\ \ldots\ \lambda y{:}t\ .\ e$$

with i occurrences of "λ". Similarly, let

$$t_1 \rightarrow^1 t_2$$

denote

$$t_1 \rightarrow t_1 \rightarrow t_1 \rightarrow \ldots \rightarrow t_2$$

with i occurrences of "→".

Note that if e has type t_2, then

$$\lambda^1 y{:}t_1\ .\ e$$

has type

$$t_1 \rightarrow^1 t_2$$

Now consider the following partially typed λ-expression E_1:

$$\begin{array}{l} \textbf{let} \\ \quad A = \Lambda t\ .\ \textbf{fix}\ a\ .\ a \\ \quad g = \Lambda t\ .\ \lambda x{:}t\ .\ \lambda y{:}s\ .\ \lambda^{k+1} y{:}C\ .\ x \\ \textbf{in} \\ \quad eq\ ?\ (g\ ?\ (A\ C))\ (A\ (s \rightarrow C \rightarrow^{k+1} C)) \end{array}$$

Based only on the "declaration" of A, we may insert any type $\alpha(t)$ for a. (We have explicitly indicated the fact that α may contain free occurrences of t.) Thus, based only on its declaration, A has a type of the form

$$\Delta t \,.\, \alpha(t)$$

From its declaration we know that g has type

$$\Delta t \,.\, t \to s \to C \to^{k\text{-}1} t$$

From the type of A we know that (A C) has type $\alpha(C)$. As a result,

$$g\ ?\ (A\ C)$$

must have type

$$(*)\quad s \to C \to^{k\text{-}1} \alpha(C)$$

The expression

$$A\ (s \to C \to^{k\text{-}1} C)$$

must have type

$$(**)\quad \alpha(s \to C \to^{k\text{-}1} C)$$

For the application of eq to be type correct, the non-type arguments must have the same type. Thus α must be chosen so that the types (*) and (**) are the same. It is not hard to convince oneself that the only acceptable choices for $\alpha(t)$ are

$$t$$

$$s \to C \to^{k\text{-}1} t$$

$$s \to C \to^{k\text{-}1} s \to C \to^{k\text{-}1} t$$

$$s \to C \to^{k\text{-}1} s \to C \to^{k\text{-}1} s \to C \to^{k\text{-}1} t$$

Thus the "possible types" of E_i are

$$s \to C \to^{k\text{-}1} C$$

$$s \to C \to^{k\text{-}1} s \to C \to^{k\text{-}1} C$$

$$s \to C \to^{k\text{-}1} s \to C \to^{k\text{-}1} s \to C \to^{k\text{-}1} C$$

Finally consider the expression E'_i defined as

$$(\Lambda s \,.\, E_i)\ C$$

It can have any type of the form

$$C \to^{ik} C$$

where k is an integer greater than 1.

Two conclusions are already apparent from this example:

1. Unlike [Mil 78], the collection of "all possible types" of a given partially typed expression does not always consist of all substitution instances of a fixed type expression (or a finite set of them).
2. The type of an expression can be much larger than the expression itself. This holds even with the graph representation of [Pat 78]. As an example, consider the expression

$$eq\ ?\ E'_{p_1}\ (eq\ ?\ E'_{p_2}\ \ldots\ (eq\ ?\ E'_{p_{n-1}}\ E'_{p_n})\ \ldots\ ))$$

where the p_i are relatively prime. The smallest acceptable type for this expression is

$$C \to^{j} C$$

where j is the product of the p_i.

A Proof

We give a reduction from the second order unification problem of [Gol 81].

We define a (unification) *term* to be either the constant C, the application of an n-ary (possibly with n = 0) function variable X_i to a sequence of n terms, or $t_1 \to t_2$, where t_1 and t_2 are terms. An *extended term* represents a function, and may mention parameter names w_i in addition to the constants and variable applications.*

The second-order unification problem is to decide whether, given two terms t_1 and t_2, there exists a substitution S for the function variables, such that $S(t_1)$ and $S(t_2)$ are identical. We view a substitution as a mapping from each n-ary function variable X_i to an extended term mentioning only $w_1, \ldots, w_n$. S is extended to apply to terms as follows:

$$S(C) = C$$

$$S(t_1 \to t_2) = S(t_1) \to S(t_2)$$

$$S(X_i(t_1, \ldots, t_n)) = S(X_i)[w_1 \leftarrow S(t_1)] \ldots [w_n \leftarrow S(t_n)]$$

where $[w_i \leftarrow t_i]$ again denotes substitution of t_i for w_i.

This problem is shown to be undecidable in [Gol 81] by a reduction from Hilbert's tenth problem**.

We define a substitution to be *proper* if each variable is mapped to an extended term mentioning only constants and parameter names; that is, if $S(X_i)$ does not contain applications of function variables. Note that for a proper substitution, each $S(X_i)$ is also a valid type expression.

If S is a solution for the second order unification problem (t_1, t_2), that is if

$$S(t_1) = S(t_2)$$

then there is a proper substitution S' with the same property; it suffices to let $S'(X_i)$ be $S(X_i)$ with each variable application replaced by the constant C.

---

*We have adopted a notation which corresponds to the syntax of types. [Gol 81] requires one function constant which we have taken to be $\to$. We choose to treat individual (first-order) variables as 0-ary function variables. Our notation for substitution also differs from that of [Gol 81].

**Hilbert's tenth problem was to obtain an algorithm for the solution of arbitrary Diophantine equations (polynomial equations over the integers). It has been shown that no such algorithm exists. See, for example, [Bel 77] for a discussion.

Goldfarb represents a natural number n as

$$C \to^{n} w_1$$

and shows that addition and multiplication are expressible.

We proceed to show that given an instance of the second order unification problem, we can effectively transform it into a partially typed λ-expression, such that the λ-expression is type correct if and only if the unification problem has a solution. Thus the existence of an algorithm to solve the type inference problem would imply a decision procedure for second order unification.

We use **let** and "proj" as in the previous section. Our general construction closely parallels the development of the preceding example.

For a given (second order) unification problem consisting of the terms t_1 and t_2, containing variables X_1, ..., X_n, we construct an expression E

let
$f_1 = v_1$
...
$f_n = v_n$
in
eq ? e_{t_1} e_{t_2}

If X_i is a variable with n_i arguments, we let v_i be

$\Lambda w_1 . \Lambda w_2 . \ldots \Lambda w_{n_i} .$
$\lambda y_1{:}w_1 . \lambda y_2{:}w_2 . \ldots \lambda y_{n_i}{:}w_{n_i} .$ **fix** $x_i . x_i$

Thus v_i must have type

$$\Delta w_1 . \Delta w_2 . \ldots \Delta w_{n_i} . w_1 \rightarrow w_2 \rightarrow \ldots \rightarrow w_{n_i} \rightarrow A_i$$

where A_i is the type associated with x_i.

Consider an assignment A of types to the above expression, which gives types of the above form to each variable f_i, and let A_i be the corresponding type of each x_i. We define such an assignment to be *proper* if and only if each A_i contains no Δ-abstractions, and no names other than C, w_1, ..., w_{n_i}. (In essence, a type assignment is proper if each A_i is an extended term of the correct arity.)

For each proper assignment A, we can construct a corresponding (proper) substitution which maps each X_i to A_i. Conversely, for each proper substitution for the X_i, we obtain a corresponding proper assignment by assigning the type $S(X_i)$ to x_i.

The expressions e_{t_1} and e_{t_2} are constructed from the unification terms t_1 and t_2 as follows. Given any term t which mentions no variables other than X_1, ..., X_n, we can construct a partially typed λ-expression e_t which, for any proper type assignment A of the above form has type $S_A(t)$. More precisely, e_t will be constructed so that there is a unique fully typed expression e' that is type correct with respect to A, such that $pe(e_t, e')$. The expression e' will have type $S_A(t)$ with respect to A.

We show how to construct e_t by induction on the structure of t.

Constants:

We take e_C to be c.

Function types:

If t has the form

$$t' \rightarrow t''$$

then we take e_t to be*

$$\lambda x . \text{proj (eq ? x } e_{t'}) \; e_{t''}$$

Applications:

If t has the form

$$X_i(t_1, \ldots, t_{n_i})$$

e_t has the form

$$f_i \; ? \; ? \; ? \; \ldots \; ? \; e_{t_1} \; e_{t_2} \; \ldots \; e_{t_{n_i}}$$

For this application to be correct, the application must be expanded to

$$f_i \; t_1 \; t_2 \; \ldots \; t_{n_i} \; e_{t_1} \; e_{t_2} \; \ldots \; e_{t_{n_i}}$$

The type of this expression is A_i with each occurrence of w_i replaced by t_i.

If the second-order unification problem has a solution, we know that there is a proper substitution which solves it. Consider this proper substitution S. Let A be the corresponding assignment of types. Thus $S = S_A$. It follows that the $S_A(t_1) = S_A(t_2)$. Thus A will result in identical types for e_{t_1} and e_{t_2}, and E will be type correct.

Conversely, if A is a proper type assignment which results in E being correctly typed, then we must have $S_A(t_1) = S_A(t_2)$ for the final application of "eq" to be type correct. Thus S_A solves the unification problem.

It remains to be shown that if E is type correct, then there is a proper assignment A which makes it type correct.

For each type τ and integer i, define $\Delta\text{-elim}_i(\tau)$ to be τ with each subexpression of the form $\Delta x . e$ replaced by e, and each identifier instance, other than free instances of w_1, ..., w_i, replaced by C. That is, each Δ-bound type variable is instantiated to C, and all extraneous free variables are replaced by C. Thus Δ-elim transforms an arbitrary type expression into an extended term of the specified arity.

Assume we are given a type assignment A to the f_i, which corresponds to the form of the v_i, and such that $T_A(e_{t_1}) = T_A(e_{t_2})$. This again determines the types A_i to be given to the x_i. Let A' be the proper assignment obtained by changing the type assigned to each x_i to be $A_i' = \Delta\text{-elim}_{n_i}(A_i)$. It is straightforward to show inductively that for each term t:

$$T_{A'}(e_t) = \Delta\text{-elim}_0(T_A(e_t))$$

It follows that A' also results in a correctly typed E.

---

*Note that t' is an arbitrary term, which may contain unification variables. Thus $\lambda x{:}t' . e_{t''}$ will not do.

Conclusions

The preceding theorem implies that full type inference for a many languages incorporating type abstraction is not possible. We made two significant assumptions. First, our language allows various forms of partial type specifications. This is certainly desirable for a real programming language. Second, we relied on the presence of a construct of completely unknown type, in this case, an obviously divergent **fix** construct. This may appear contrived, but it is very difficult to disallow such "obviously divergent" constructs in a real programming language. Furthermore, other constructs, such as free identifiers of unknown type*, or an identifier of type Δt . t can serve the same role in the proof.

For a language such as Russell, the statement of the theorem becomes slightly different. The following three kinds of type omissions must be allowed for a similar proof to go through. Unfortunately they are all nearly essential in practice:

1. Even in our fully typed λ-calculus, result types for λ-abstractions were implicit. In a language with Pascal like syntax we must explicitly allow their omission. Disallowing this omission makes it very clumsy to write higher-order functions, or functions producing structured values.
2. It is not necessary to allow omission of parameter types. It must however be possible to omit types of declared identifiers. (This is allowed even in certain Pascal declarations.) Since declarations may be recursive, there is no **fix** construct in the language.
3. Functions may take multiple arguments, some of which may be types. It must be possible to omit type arguments from an argument list.

This leaves open at least two questions. First, it is unclear whether the problem becomes decidable if we disallow partial specification. Second, but of greater practical interest, is the issue of usable approximation algorithms, and (cleanly defined) solvable subproblems. [Boe 85b] gives a rather weak and very ad-hoc approximation algorithm. This makes it difficult to give any guarantees to the programmer about when type inference will succeed. Other implementations of comparable languages are either significantly more restrictive, as in [Mil 78], or share at least some of these problems. [McC 84] proposes an approximation algorithm, but still leaves open a number of associated practical problems.

*It suffices to replace each occurrence of fix x_i . x_i in the construction of E by

$$x_i\, w_1 \ldots w_{n_i}$$

where x_i is a free identifier of unknown type.

Acknowledgement

The author is grateful to John Mitchell who first made him aware of the result in [Gol 81] and showed that it was equivalent to a different formulation of the problem, to which we had reduced a version of the Russell type inference problem. He also pointed out the need for prohibiting arbitrary placement of type applications in the statement of the preceding theorem.

References

[Bel 77] J. L. Bell and M. Machover, *A Course in Mathematical Logic*, North Holland, 1977, pp. 284-314. The result is due to Matijasevic.

[Boe 80] H. Boehm, A. J. Demers, and J. E. Donahue, "An Informal Description of Russell", Department of Computer Science, Cornell University, Technical Report 80-430, 1980.

[Boe 85a] H. Boehm, A. J. Demers, and J. E. Donahue, "A Programmer's Introduction to Russell", Department of Computer Science, Rice University, Technical Report 85-16, 1985.

[Boe 85b] H. Boehm, and A. J. Demers, "Implementing Russell", Department of Computer Science, Rice University, Technical Report 85-25, 1985.

[Gol 81] W. D. Goldfarb, "The Undecidability of the Second-Order Unification Problem", Theoretical Computer Science 13 (1981), pp.225-230.

[Gor 79] M. J. Gordon, R. Milner, and C. P. Wadsworth, *Lecture Notes in Computer Science, Vol. 78: Edinburgh LCF*, Springer Verlag, 1979.

[Jen 84] R. D. Jenks, "A Primer: 11 Keys to the new Scratchpad", Proceeding of EUROSAM '84, Springer Lecture Notes in Computer Science # 174, pp. 123-147.

[Lei 83] D. Leivant, "Polymorphic Type Inference", Proceedings, 10th Annual ACM Symposium on Principles of Programming Languages, 1983, pp. 88-98.

[McC 84] N. McCracken, "The Typechecking of Programs with Implicit Type Structure", Lecture Notes in Computer Science, Vol 173: Proceedings of the International Symposium on Semantics of Data Types, Springer Verlag, 1984, pp. 301-315.

[Mee 83] L. Meertens, "Incremental Polymorphic Typechecking in B", Proceedings, 10th Annual ACM Symposium on Principles of Programming Languages, 1984, pp. 265-275.

[Mil 78] R. Milner, "A Theory of Type Polymorphism in Programming", Journal of Computer and System Sciences 17 (1978), pp. 348-375.

[Mit 84] J. C. Mitchell, "Type Inference and Type Containment", Lecture Notes in Computer Science, Vol 173: Proceedings of the International Symposium on Semantics of Data Types, Springer Verlag, 1984, pp. 257-277.

[MQu 84] D. MacQueen, G. Plotkin, and R. Sethi, "An Ideal Model for Recursive Polymorphic Types", Proceedings, 11th Annual ACM Symposium on Principles of Programming Languages, 1984, pp. 165-174.

[Pat 78] M. S. Paterson, and M. N. Wegman, "Linear Unification", J. Comput. Syst. Sci. 16, 2 (April 1978), pp. 158-167.

[Rey 74] J. Reynolds, "Towards a Theory of Type Structure", Paris Colloquium on Programming, 1974, pp. 408-424.

[Wie 83] R. Wiener and R. Sincovec, *Programming in Ada*, John Wiley & Sons, New York, 1983.