On the Complexity of ML$^+$ Type Inference

Fritz Henglein*
Courant Institute

Kenneth J. Perry†
IBM T.J. Watson Research Center

Abstract

The programming language ML$^+$ extends the ML language with a polymorphic fix-point construct. In this paper, we clarify the combinatorial properties of performing type inference in this language. Specifically, we demonstrate an EXPTIME lower bound on recognizing typable ML$^+$expressions. We also show that this problem can be reduced to the simpler problem of recognizing typable expressions in a proper subset of ML$^+$. Finally, we show that the type inference problem is equivalent to the problems of semi-unification and polymorphic unification.

1 Introduction

One reason for the popularity of the ML programming language [Mil78] is its type inference system. This system frees the programmer from having to make explicit type declarations. Instead, all typing information can be inferred from the program's structure. A second reason is a polymorphic `let` construct that facilitates code-sharing. For example, one can specify a function that sorts arrays of *any* type of element. Contrast this to most languages in which it is necessary to have two separate, nearly identical routines to sort integer and floating-point numbers.

When limitations on the utility of the `let` construct became apparant, the ML$^+$ language [Myc84] [KTU88b] was proposed as a proper

*Courant Institute of Mathematical Sciences, New York University, 715 Broadway, 7th Floor, New York, NY 10012. Internet: henglein@nyu.edu

†IBM Research, PO BOX 704, Yorktown Heights, NY 10598. Internet: kjp@ibm.com

extension of ML. An example (from [Myc84]) will motivate this extension.

Consider the following simultaneous definition of functions *map* and *squarelist*:

$$\begin{aligned} map(f,l) &= \texttt{if } null(l) \texttt{ then } nil \texttt{ else} \\ &\quad cons(\ f(hd\ l), map(f, tl\ l)\ ) \\ squarelist(l) &= map(\ \lambda x.\, x * x,\ l\ ) \end{aligned}$$

By the type inference rules of ML, the functions are determined to have *monomorphic* types

$map : (INT \rightarrow INT) \times INT\text{list} \rightarrow INT\text{list}$

$squarelist : INT\text{list} \rightarrow INT\text{list}$

However, if the functions were defined sequentially, the expected *polymorphic* type would be inferred for *map*, namely

$$map : \forall\alpha\beta.(\alpha \rightarrow \beta) \times \alpha\text{list} \rightarrow \beta\text{list}$$

The failure to derive the expected polymorphic type for *map* would be even more serious if we simultaneously defined a third function that applied *map* to a $BOOL$list. In this case, correct typing is not possible since the inference rules would require that the distinct types INT and $BOOL$ be equal.

The contribution of ML$^+$ was a polymorphic `fix` construct that permitted mutually recursive definitions. Using this construct to define the two functions as

$$\texttt{fix}\ (map, squarelist).(\lambda(f,l)\ldots., \lambda l\ldots.)$$

yields the expected polymorphic typing for *map*.

One question that is raised by polymorphism is its computational complexity. Experience with ML had led many to believe that polymorphic type inference was not a difficult problem. In fact, a "folk theorem" developed that claimed a linear time algorithm for performing type inference. It was not until the recent results of Kanellakis and Mitchell [KM89] that this notion was debunked. To the contrary, [KM89] demonstrated the PSPACE-hardness of recognizing typable ML expressions. Our goal in this paper is to answer the same question for the language ML$^+$.

1.1 Related work and previous results

The expressions of ML$^+$considered in this paper are given by

$$E ::= x \mid (EE) \mid \lambda x.E \mid \texttt{let}\ x = E\ \texttt{in}\ E \mid \texttt{fix}\ x.E$$

where x ranges over a given set of variables. The typing rules for ML$^+$ (see appendix) are identical to the ML typing rules [DM82] except for a more general rule for `fix`-expressions.

Polymorphism is introduced by interpretting each occurrence of x in the `let` and `fix` constructs as being of different types. ML$^+$properly extends the core ML language in its inclusion of the `fix`construct.

The type of an expression that excludes `let` and `fix` may be inferred in linear time by simple unification. Until recently, it was believed that this was also true for expressions including `let`. Kanellakis and Mitchell [KM89] reduced the problem of recognizing typable ML expressions to a "polymorphic unification" problem and demonstrated its PSPACE-hardness. However, they were only able to give an EXPTIME upper bound.

Mycroft and others [Myc84] [Mee83] [KTU88b] recognized the practical limits of having `let` as the sole means of polymorphism and extended ML with the polymorphic `fix`, thereby creating ML$^+$. Kfoury et. al. [KTU88a] formally established that this extension increases the language's expressiveness but bounds on the type inference problem for this language were left open.

Recently, several people have shown that the type inference problem for ML$^+$can be reduced to the problem (called "semi-unification" in [Hen88]) of solving systems of inequalities on type expressions [Mit88] [KTU88b] [Hen88][GR88]. Again, the complexity of solving these systems was left open (although there are partial results in [Cho86] and [PM88].)

2 Main results and significance

In this paper, we address the problems of type-inference in ML$^+$, semi-unification, and extended polymorphic unification. The contributions of this paper are in demonstrating

- The equivalence of ML$^+$ type inference, semi-unification, and extended polymorphic unification.

- The above problems may be reduced to a simpler problem, namely recognizing typable expressions in the proper subset of ML^+ containing no `let` expressions and at most one `fix`.

  Mycroft [Myc84] conjectured that the nesting of `fix` constructs might have to be limited on pragmatic grounds "due to the exponential cost of analyzing nested `fix` definitions". Our result shows that the essential combinatorial nature of the inference problem is already inherent in a single `fix` construct. (However, from the point of program expressiveness, multiple `fix` constructs might still be needed.)

- A lower bound of EXPTIME on these problems.

  This clarifies the combinatorial nature of the problem and perhaps explains why algorithms for these problems have not been found.

3 Problem definitions

In this section we define each of the problems considered. We shall show their log-space equivalence in the following section.

Typability of ML^+ *expressions* is the problem of determining whether a type can be derived for an arbitrary ML^+ expression using the language's type inference system (given in the appendix).

Semi-unification is a problem akin to unification. The preordering $\leq$ of *subsumption* on first-order terms is defined by $M \leq N$ if there exists a substitution σ such that $\sigma(M) = N$. A system

$$\{M_{11} = M_{12}, \ldots, M_{k1} = M_{k2} \\ N_{11} \leq N_{12}, \ldots, N_{l1} \leq N_{l2}\}$$

of term equations and term subsumption inequalities is *semi-unifiable* if there is a substitution σ such that all of the following equalities and subsumption statements hold:

$\sigma(M_{11}) = \sigma(M_{12}), \quad \ldots, \quad \sigma(M_{k1}) = \sigma(M_{k2}),$
$\sigma(N_{11}) \leq \sigma(N_{12}), \quad \ldots, \quad \sigma(N_{l1}) \leq \sigma(N_{l2})$

Polymorphic unification, an extension of ordinary unification, which was recently introduced by Kanellakis and Mitchell[KM89], defines a sub-class of semi-unification problems. For example, if $M_1[x, \ldots, x]$ is a term with k occurrences of x and if M_2, M_3 are other terms, then the

extended terms M_3 and ($\texttt{let}\ x = M_2\ \texttt{in}\ M_1[x,\ldots,x]$) are unifiable if and only if the system

$$\{\ M_1[x_1,\ldots,x_k] = M_3, x = M_2,\ x \leq x_1,\ldots,x \leq x_k\}$$

is semi-unifiable.

4 Equivalence Results

The main result of this section establishes the equivalence of severalof these problems and is proved in the following.

Theorem 1 *The following three problems are log-space equivalent.*

1. *Typability of arbitrary* ML$^+$ *programs;*
2. *typability of* ML$^+$ *programs of the form* `fix` $x.E$ *where* E *is* `let`*- and* `fix`*-free;*
3. *semi-unifiability of arbitrary systems of term equations and subsumption inequalities.*

This theorem shows that

1. type checking for ML$^+$ programs with only a single `fix` and no `let` is already PSPACE-hard;
2. nesting of `fix` expressions does not make type checking harder (this is in contrast to Mycroft's statement in [Myc84]);
3. `fix` expressions are at least as expensive as `let` expressions as far as type checking is concerned;
4. semi-unification captures the combinatorial essence of type checking ML$^+$ programs.

Proof (Sketch)

We sketch a proof of the two reductions (1) $\Rightarrow$ (3) and (3) $\Rightarrow$ (2). The first of these reductions can be found in [Hen88]. We shall briefly reiterate the outline of that reduction. In the first step we label the nodes of the syntax tree of a given ML$^+$ program with distinct type variables. We then collect a set of equations between these type variables and type expressions of the form $\tau_1 \rightarrow \tau_2$ from the typing rules (ABS) and (APPL). For every λ-bound variable x, labelled with type variable t, and every occurrence of x, labelled with

t', we add the equation $t = t'$. Now, for every `let`- and `fix`-bound variable x, labelled with the type variable t, and every occurrence of that x, labelled with t', we collect all the λ-bound variables and their type labels $t_1, \ldots, t_k$ in whose scope x occurs and add the subsumption inequality $f(t, t_1, \ldots, t_k) \leq f(t', t_1, \ldots, t_k)$; here f is any suitable function symbol. The resulting system of equations and inequalities has the property that it is semi-unifiable if and only if the original ML$^+$ program is typable.

For the second reduction, (3) $\Rightarrow$ (2), let

$$\{ \begin{array}{lllllll} M_{11} & = & M_{12}, \ldots, M_{k1} & = & M_{k2}, \\ N_{11} & \leq & N_{12}, \ldots, N_{l1} & \leq & N_{l2} \} \end{array}$$

be a system of equations and inequalities with variables $x_1, \ldots, x_k$. From [KM89] we know that every term can be encoded by `fix`- and `let`-free λ-expressions and that there is a λ-expression "=" that encodes equality between terms. Similarly, tuples $[L_1, \ldots, L_h]$ and tuple selection functions $\bar{i}([L_1, \ldots, L_h]) = L_i$ can be represented by standard constructions. Now, the λ-expression

$$\texttt{fix}\ f.\ \ \lambda x_1 \ldots x_k.\ K[M_1, \ldots, M_k][\lambda y_1 \ldots y_k.\bar{1}(f y_1 \ldots y_k) = N_1, \ldots, \lambda y_1 \ldots y_k.\bar{k}(f y_1 \ldots y_k) = N_k]$$

is typable if and only if the original system of equations and inequalities is semi-unifiable.

5 Lower Bound

The main result of this section is the following.

Theorem 1 *Recognizing typable* ML$^+$ *expressions is hard for EXPTIME.*

By the equivalence results of the previous section, this demonstrates an EXPTIME lower bound for the semi-unification and polymorphic unification problems as well.

This result contrasts with the PSPACE lower bound and EXPTIME upper bound derived by [KM89] for recognizing typable ML expressions. Using `let` constructs nested to a depth of n [KM89] can only specify trees of depth n. With a single, un-nested `fix` we are able to specify trees of depth 2^n. Thus, the `fix` construct seems to be much more powerful a means of introducing polymorphism than is the `let` construct.

Proof: (sketch)

Given a PSPACE-bounded Alternating Turing Machine[1] M with input x, we use the techniques of [CKS81] to deterministically simulate its behavior. Our contribution is in showing that this simulation is possible by the extended polymorphic unification of a graph[2] G whose size is polynomial in the description of M and x and that represents the type of a `fix` expression.

The simulation of [CKS81] organizes the configurations of M on x as nodes of a "computation tree" and assigns each a value such that M accepts if and only if the root is assigned **true**[3]. But, since M is PSPACE-bounded, the depth of this tree can be as great as 2^n, where n is the length of x. Therefore, the crux of our proof is in concisely representing trees of depth 2^n as the type of a `fix` expression.

The graph of Figure 1 compactly respresents a binary tree whose nodes are instances of the graph contained in the circle (the "super-node"). This super-node will contain many sub-graphs related to the simulation, such as an encodings of configurations, Boolean functions, and a "next-configuration" function that maps a configuration into its successor configurations according to the next-state function of M. We first concentrate on specifying the sub-graph that keeps count of a node's depth with the computation tree.

Throughout our simulation, a Boolean variable v will be represented by four nodes $v_0, \ldots, v_3$. If $v_0 = v_1$ (i.e., v_0 and v_1 unify), then we interpret v as having value **true** or 1; if instead $v_2 = v_3$, we intepret v as having value **false** or 0. A single bit of the counter is shown in Figure 2. It takes a bit i and a "gate" value g as inputs, and produces o as output according to the rule $o = (i + g) \bmod 2$. That is, the graph counts up when the gate value is true.

Now consider connecting n of these counters in series (with inputs $i^0, \ldots, i^{n-1}$, outputs $o^0, \ldots, o^{n-1}$, and gates $g^0, \ldots, g^{n-1}$) according to the following equations:

---

[1]Recall, EXPTIME = Alternating PSPACE.

[2]Type expressions have a natural representation as graphs, which we use here for convenience. It is equally possible to directly give an equivalent system of constraints on type expressions.

[3]The root of this tree is the initial configuration and the children of a node are its immediate successor configurations. Leaves are assigned Boolean values indicating whether they are accepting configurations. Nodes that are universal (resp., existential) configurations are assigned a value that is the conjuction (resp., disjunction) of the values assigned to its children.

$$
\begin{aligned}
g^0 &= e \\
g^{j+1} &= e \wedge i^j
\end{aligned}
$$

Then letting I denote the integer represented by the Boolean string $i^{n-1}i^{n-2}\ldots i^0$ and O the integer with representation $o^{n-1}i^{n-2}\ldots o^0$, we can prove that $O = (I + e) \bmod n$. That is, setting e to 1 causes the n-bit counter to be incremented.

Having shown that trees of depth 2^n are constructible, we can adapt the techniques of [DKM84] [DKS88] that give encodings of Boolean functions as graphs. We can then embed a "next-configuration" function within the super-node to ensure that the resulting tree is in fact the computation tree of M on input x. The technique of [KM89] that computes values for nodes in a bottom-up manner can also be adapted to assign values to the nodes of this "computation-tree". Finally, the results of [KM89] can be extended so as to extract an ML^+ expression from our graph. ∎

References

[Cho86] C.-T. Chou. *Relaxation Proceses: Theory, Case Studies and Applications.* PhD thesis, University of California at Los Angeles, 1986.

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the ACM*, 287(1):114–133, 1981.

[DKM84] C. Dwork, P. Kanellakis, and J. C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1:35–50, 1984.

[DKS88] C. Dwork, P. Kanellakis, and L. Stockmeyer. Parallel algorithms for term matching. *SIAM Journal of Computing*, 17(4):711–731, 1988.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[GR88] P. Giannini and S. Ronchi Della Rocca. Characterization of typings in polymorphic type discipline. In *Proc. Third Annual Symposium on Logic in Computer Science*, pages 61–71, IEEE, 1988.

[Hen88] F. Henglein. Type inference and semi-unification. In *Proc. ACM Symposium on LISP and Functional Programming*, page ?, ACM, 1988.

[KM89] P.C. Kanellakis and J.C. Mitchell. Polymorphic unification and ML typing (extended abstract). In *Proc. Sixteenth ACM Symposium on Principles of Programming Languages*, ACM, January 1989.

[KTU88a] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. ?? In *Proc. Third Annual Symposium on Logic in Computer Science*, page ??, ?, 1988.

[KTU88b] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. A proper extension of ml with effective type assignment. In *Proc. Fifteenth ACM Symposium on Principles of Programming Languages*, pages 58–69, ACM, 1988.

[Mee83] L. Meerteens. Incremental polymorphic type checking in B. In *Proc. Tenth ACM Symposium on Principles of Programming Languages*, pages 265–275, ACM, 1983.

[Mil78] R. Milner. A theory of polymorphism in programming. *JCSS*, 17:348–375, 1978.

[Mit88] J.C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2/3):, 1988.

[Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proc. International Symposium on Programming, Lecture Notes in Computer Science 167*, pages 217–228, 1984.

[PM88] D.S. Parker and R.R. Muntz. A theory of directed logic programs and streams. In *Proc. Fifth Internation Conference on Logic Programming*, pages 620–650, MIT Press, 1988.

A ML$^+$ Typing Rules

ML$^+$ is an extended λ-calculus. The type expressions are given by

$\tau := t \mid \tau \rightarrow \tau$

$t :=$ (type variables)

$\sigma := \tau \mid \forall t.\sigma$

Type expressions derived from τ above are called *monotypes* and the larger set of type expressions derived from σ are *polytypes.* A type assignment is a mapping from λ-calculus variables to type expressions. For detailed definitions of λ-expressions, type expressions, and type assignments we refer to [DM82] and [Myc84] or any number of other papers on type theory.

The canonical type inference system for the ML$^+$ [Myc84] is given below. Let A range over type assignments, x over λ-calculus variables, t over type variables, e and e' over expressions, τ and τ' over monotypes, and σ and σ' over polytypes.

(TAUT) $A\{x:\sigma\} \supset x:\sigma$

(INST) $$\frac{A \supset e:\forall t.\sigma}{A \supset e:\sigma[\tau/t]}$$

(GEN) $$\frac{A \supset e:\sigma \qquad (t \text{ not free in } A)}{A \supset e:\forall t.\sigma}$$

(APPL) $$\frac{A \supset e:\tau' \to \tau \qquad A \supset e':\tau'}{A \supset (ee'):\tau}$$

(ABS) $$\frac{A\{x:\tau'\} \supset e:\tau}{A \supset \lambda x.e:\tau' \to \tau}$$

(LET) $$\frac{A \supset e:\sigma \qquad A\{x:\sigma\} \supset e':\sigma'}{A \supset \mathtt{let}\ x = e\ \mathtt{in}\ e':\sigma'}$$

(FIX-P) $$\frac{A\{x:\sigma\} \supset e:\sigma}{A \supset \mathtt{fix}\ x.e:\sigma}$$

B Figures

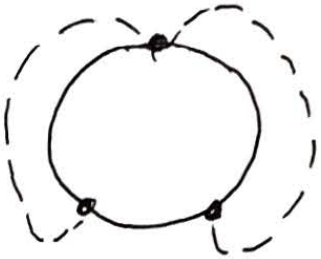
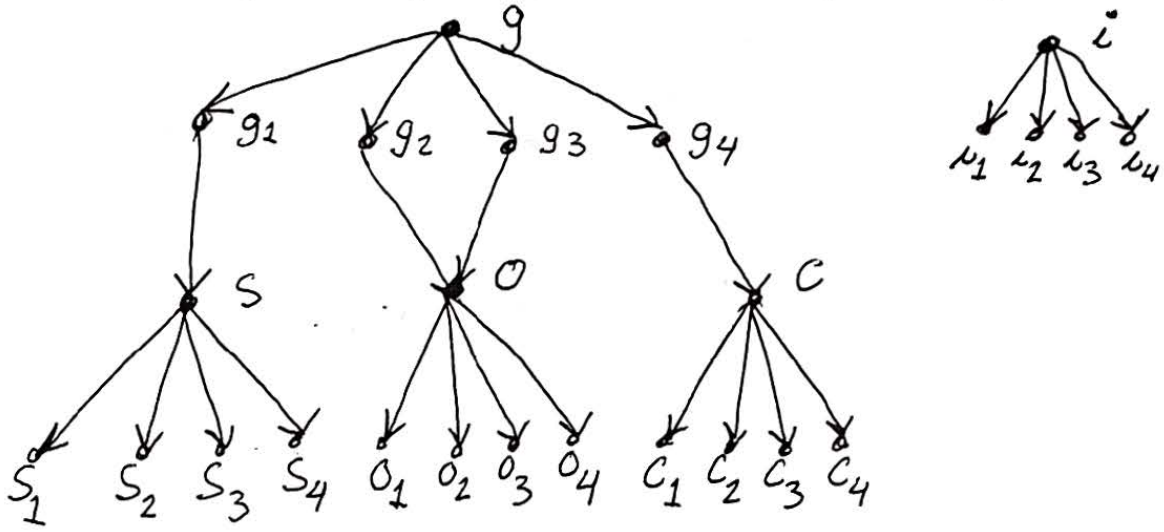


Figure 1: Compact respresentation of a binary tree of "super-nodes"



The following relations hold among the nodes:

$c_j = i_j \qquad j = 0, \ldots, 3$

$s_j = i_{(j+2) \bmod 4} \quad j = 0, \ldots, 3$

Figure 2: A single bit of the n-bit counter