A Logical Algorithm for ML Type Inference

David McAllester

Toyota Technological Institute at Chicago
mcallester@tti-c.org

Abstract. This paper gives a bottom-up logic programming formulation of the Hindley-Milner polymorphic type inference algorithm. We show that for programs of bounded order and arity the given algorithm runs in $O(n\alpha(n) + dn)$ time where n is the length of the program, d is the "scheme depth" of the program, and α is the inverse of Ackermann's function. It is argued that for practical programs d will not exceed 5 even for programs with hundreds of module layers. This formulation of the Hindley-Milner algorithm is intended as a case study in "logical algorithms", i.e., algorithms presented and analyzed as bottom-up inference rules.

1 Introduction

This paper is a case study in the use of bottom-up logic programming for the presentation and analysis of complex algorithms. The use of bottom-up logic programming for algorithm presentation has been developed in a recent series of papers whose main contributions are theorems governing the running time of these programs [6, 1, 2]. This paper explores the use of these run time theorems for the presentation and analysis of the Hindley-Milner type inference algorithm used in the ML and Caml programming languages [7]. It is known that for programs where procedures can take an unbounded number of arguments and be of unboundedly high order (procedures which take procedures which take procedures ...) the type inference problem is complete for exponential time [4, 5]. In practice human written procedures never take more than twenty arguments and are never more than fifth order. It is known that for bounded order and arity the type inference problem can be done in polynomial time [3]. However, I am not aware of a published analysis giving a particular polynomial running time for this problem. This paper gives an inference rule presentation of a version of the Hindley-Milner type inference algorithm and shows that this version runs in time $O(n\alpha(n) + dn)$ where d is the "scheme depth" of the program. The algorithm presented here is very similar to the one described in [8] although no formal analysis of running time is given there. Section 5 defines scheme depth and argues that it can be expected to be less than 5 even for programs with hundreds of module layers.

2 The LDP Programming Language

Here we use (essentially) the logic programming language and run time model specified in [2]. For convenience we will here refer to this programming language as LDP for Logic programs with Deletions and Priorities. It is important to note that deletions and priorities have been widely discussed in the logic programming literature. The main contribution of [2] is the definition of an abstract notion of running time and a proof that this abstract definition of running time can be implemented on a conventional random access computer. This run time result is stated in section 3. We now define an LDP program to be a set of inference rules where an inference rule is specified by the category r in the following grammar.

$$\begin{aligned}
N &\equiv i \mid n \mid N_1 + N_2 \mid N_1 * N_2 \\
H &\equiv x \mid f(\tau_1, \ldots, \tau_k) \\
\tau &\equiv H \mid N \\
A &\equiv P(\tau_1, \ldots, \tau_k) \mid N_1 \leq N_2 \\
C &\equiv P(\tau_1, \ldots, \tau_k) \mid \mathrm{del}(P(\tau_1, \ldots, \tau_k))
\end{aligned}$$

$r \equiv A_1, \ldots, A_n \rightarrow C_1, \ldots, C_k$ with priority N

Every variable in the priority N occurs in A_1.

If $A_i = P(t_1, \ldots t_k)$ then A_i does not contain $+$ or $*$.

If $A_i = N_1 \leq N_2$ then every variable in A_i occurs in some A_j for $j < i$.

Every variable in C_i occurs in some A_j.

This is two-sorted grammar with a sort for Herbrand terms (H) and a separate sort for integers (N). There are two sorts of variables — Herbrand variables such as x in the grammar for H and integer variables such as i in the grammar for N. In the grammar for integers, n ranges over any integer constant. We allow predicates and functions to take either sort as arguments although we assume that each predicate and function symbol has a specified arity and a specified sort for each argument and all applications of predicate and function symbols must respect this specification to be well formed. The function symbol f in the grammar for Herbrand terms should be viewed as a data constructor.

The inference rules are to be run bottom-up. For rules not involving deletion, a state of the system is a database of assertions which grows monotonically by adding new assertions derivable from some rule and assertions already in the database. When deletion is involved each rule states that its conclusion actions are to be taken taken whenever its antecedents are satisfied. The actions are either additions to, or removals from, a database of assertions of the form $P(\tau_1, \ldots, \tau_n)$. A particular use (instance) of a rule involves particular values for all the variables of that rule. Given values for the variables in the first antecedent, a rule is associated with an integer priority where the integer 1 is the highest priority (priority 1 rules run first). Priorities smaller than 1 are treated as equivalent to 1 but (instances of) rules with larger priority values run only if no higher priority rule instance can run.

Figure 1, taken from [2], gives an LDP implementation of the Dijkstra shortest path algorithm. The input is an initial database containing assertions of the form $E(v, c, u)$ stating that there is an edge from v to u of cost c plus an assertion of the form $\mathsf{source}(v)$ stating that v is the source node. In the figure each rule is given a name and a priority. For example, rule D2 has priority 1 while rule D3 has priority $d+2$ where the integer variable d occurs in the first antecedent of the rule. Different uses of the rule D3 run at different priorities. This has the effect that distances are established in a shortest-first order as in the normal Dijkstra shortest path algorithm. The general run time model for inference rules given in section 3 implies that these rules run in $O(e \log e)$ where e is the number of initial assertions. Note that $d' < d$ can be written as $d' + 1 \leq d$.

We now consider the operational semantics an arbitrary set of LDP rules in more detail. A ground assertion is one not containing variables or arithmetic operations. A program state is a set S of ground assertions of the form $P(t_1, \ldots, t_k)$ and deletion assertions of the form $\mathrm{del}(P(t_1, \ldots, t_k))$. Deletion actually adds an assertion of the form $\mathrm{del}(P(t_1, \ldots, t_k))$ so that the set D grows monotonically over the execution of the algorithm. However, only the positive elements of D that have not been deleted are "visible" to the antecedents of the inference rules. Once a ground assertion has been both asserted and deleted new assertions or deletions of that ground assertion do not change the state of the database. Let r be the rule $A_1, \ldots, A_n \rightarrow C_1, \ldots, C_m$ at priority N. Let σ be a substitution assigning (ground) values to all the variables in the rule r. The pair $\langle r, \sigma\rangle$ will be called an instance of r. The instance $\langle r, \sigma\rangle$ is called *pending* at state S if each antecedent $\sigma(A_i)$ holds in the state S and firing the rule would change the state, i.e., $S \neq S \cup \{\sigma(C_1), \ldots, \sigma(C_m)\}$. We say that state S can make an R-transition to state S', written $S \stackrel{R}{\rightarrow} S'$, if there exists a pending rule instance $\langle r, \sigma\rangle$ with $r \in R$ such that no pending stance of a rule in R has higher priority and S' is $S \cup \{\sigma(C_1), \ldots, \sigma(C_m)\}$ where $C_1, \ldots, C_m$ is the conclusion of r. We say that W is a final state for R on input S if no rule instances are pending in W and W is reachable from S by a sequence of R-transitions. One should think of R as a (don't care) nondeterministic procedure for mapping input S to output W.

$$\texttt{(D1,1)}\ \frac{\mathsf{source}(v)}{\mathsf{dist}(v,0)} \qquad \texttt{(D2,1)}\ \frac{\begin{array}{l}\mathsf{dist}(v,d)\\ \mathsf{dist}(v,d')\\ d' < d\end{array}}{\mathrm{del}(\mathsf{dist}(v,d))} \qquad \texttt{(D3,}d\texttt{+2)}\ \frac{\begin{array}{l}\mathsf{dist}(v,d)\\ E(v,c,u)\end{array}}{\mathsf{dist}(u,d+c)}$$

Fig. 1. Dijkstra Shortest Path.

3 The Running Time of LDP Programs

We say that R terminates on input S if there is no infinite sequence S_1, S_2, S_3, ... such that $S = S_1$ and $S_i \stackrel{R}{\to} S_{i+1}$. The Dijkstra shortest path algorithm in figure 1 terminates even if we assign all rules priority 1 so that they can fire in any order. However, we are interested here in much more refined analysis of running time when the rule set is implemented on a conventional random access computer. To make statements about conventional running time we introduce the notion of a prefix firing. A *prefix instance* of r with n antecedents is a triple $\langle r,\ i,\ \sigma\rangle$ where $1 \leq i \leq n$ and σ is a ground substitution defined on (only) the variables occurring in the antecedent prefix $A_1,\ \ldots,\ A_i$ of r. Intuitively, a prefix instance is a way of instantiating the first i antecedents of the rule. Let W be a final state of input S for rule set R. An atomic formula $P(t_1,\ \ldots,\ t_k)$ in W will be said to have been asserted and will be said to have been deleted if W contains $\mathrm{del}(P(t_1,\ \ldots,\ t_k))$. An assertion is visible to the antecedents of rules if it has been asserted but has not been deleted.

> **Definition:** A *prefix firing* of a rule r over a final state W is a prefix instance $\langle r,\ i,\ \sigma\rangle$ of r such that $\sigma(A_1), \ldots, \sigma(A_i)$ have all been asserted (and possibly also deleted) where $A_1, \ldots, A_i$ are the first i antecedents of r.

The basic result in [6] is that for a rule set R in which all rules have priority 1 (and can therefore fire in any order), and where no rule involves deletion, the (unique) final state can be computed time proportional to the number of prefix firings over the final state. This bound also holds for rules with fixed priorities (no variables in the priority expressions) and deletions. However the run time bound from this "naive" count of prefix firings is typically quite loose for rule sets involving deletions and priorities. Rather than count all prefix firings we want to only count the "visible" prefix firings. To define the visible prefix firings we consider computation histories. A complete R-computation from S is a sequence $S_1, S_2, \ldots, S_T$ such that $S_1 = S$, $S_i \stackrel{R}{\to} S_{i+1}$ and S_T is a final value of S, i.e., there are no instances of rules in R that are pending in S_T.

> **Definition:** A state S is said to be visible to a prefix instance $\langle r,\ i,\ \sigma\rangle$ if no pending instance of a rule in R has priority higher than the priority of $\langle r,\ i,\ \sigma\rangle$. A prefix firing $\langle r,\ i,\ \sigma\rangle$ over the final state of a computation is said to be visible in a computation if either $i = 1$ and r is variable priority or the computation contains a state S visible to $\langle r,\ i,\ \sigma\rangle$ such that $\sigma(A_1), \ldots \sigma(A_i)$ all hold in S (have been asserted but not deleted) where $A_1, \ldots, A_i$ are the first i antecedents of r.

Every visible prefix firing is a prefix firing over the final state, but the visible prefix firings are usually only a small fraction of the prefix firings. A first-antecedent prefix firing is just a prefix firing $\langle r, i, \sigma\rangle$ with $i = 1$. The following is the main theorem of [2].

Theorem 1. *If the rule set R terminates on initial state S then a complete R-computation from S can be computed on a RAM machine extended with constant time hash table operations in time proportional to $|S| + P_1 + P_2 \log N$ where $|S|$ is the number of assertions in S, P_1 is the number of visible prefix firings of fixed priority rules; P_2 is the number of visible prefix firings of variable priority rules; and N is the number of distinct priorities of first-antecedent prefix firings of variable priority rules.*

To see how this theorem can be applied we now give a running time analysis of the Dijkstra shortest path algorithm shown in Figure 1. Note that for a state to be visible to (an instance of) a rule there must not be any pending higher priority rule. States where no high-priority rule is pending often satisfy certain invariants. The priority of a state is the priority of a highest-priority (lowest integer) pending rule instance (or $+\infty$ if there are no pending rule instances). The rule D2 ensures that in any state with priority 2 or larger we have at most one distance associated with each node. So this invariant holds in any state visible to the rule D3. The priority of an instance of rule D3 is determined by the value of the integer variable d appearing in the first antecedent — shorter distances get higher priority. Note that when an instance of D3 runs all pending rule firings involve priorities at least as large as d. Since an invocation of rule D3 must generate a bound at least as large as d, all future bounds will be at least as large as d. Hence, in any state with priority $d+2$ or greater, any visible assertion of the form $\mathsf{dist}(u,d)$ must have the property that d is the actual shortest distance to the node u. Hence, for a given node u there is only one value of d used in a full firing of rule D3. This implies that the total number of firings of the first two antecedents of D3 is at most the number of edges in the graph. Each firing of D2 occurs in a state immediately following a firing of D3 and for each firing of D3 there is at most one firing of D2. Hence the total number of firings of D2 is also at most the total number of edges. Furthermore, this implies that the total number of assertions of the form $\mathsf{dist}(u,d)$ ever asserted is at most e. All of these "ever-asserted" assertions must be included as prefix firings of the first antecedent of the variable priority rule D3. However, for the two antecedent prefix firings of D3 we need only consider a single value of d for each value of u. Hence the total number of prefix firings is $O(e)$. The number of distinct priorities associated with rule D3 is at most e. So the abstract run time of rule D3 is $O(e \log e)$. The run time of D3 dominates and the algorithm runs in $O(e \log e)$ time.

4 Union-Find

Figure 2, taken from [2], gives an LDP implementation of the union-find algorithm. The algorithm is not particularly elegant, but it does show that LDP with its associated abstraction notion of run time is sufficiently powerful to express a useful implementation of union-find. Note that these rules all have priority either 1 or 2. The rules take as input assertions of the form $\mathsf{union}(x,y)$ and produce as output assertions of the form $\mathsf{find}(x,f)$. Any state in which no union-find rules

$$\texttt{(UF1,1)}\quad \frac{\mathsf{union}(x,y)}{\begin{array}{l}\mathsf{nofind}(x)\\ \mathsf{nofind}(y)\end{array}} \qquad \texttt{(UF2,1)}\quad \frac{\mathsf{find}(x,y)}{\mathrm{del}(\mathsf{nofind}(x))} \qquad \texttt{(UF3,1)}\quad \frac{\begin{array}{l}\mathsf{find}(x,y)\\ \mathsf{find.arc}(y,z)\end{array}}{\begin{array}{l}\mathsf{find}(x,z)\\ \mathrm{del}(\mathsf{find}(x,y))\end{array}}$$

$$\texttt{(UF4,1)}\quad \frac{\begin{array}{l}\mathsf{union}(x,y)\\ \mathsf{find}(x,z)\\ \mathsf{find}(y,z)\end{array}}{\mathrm{del}(\mathsf{union}(x,y))} \qquad \texttt{(UF5,2)}\quad \frac{\mathsf{nofind}(x)}{\begin{array}{l}\mathsf{find}(x,x)\\ \mathsf{size}(x,1)\end{array}} \qquad \texttt{(UF6,2)}\quad \frac{\begin{array}{l}\mathsf{union}(x,y)\\ \mathsf{find}(x,x')\\ \mathsf{find}(y,y')\end{array}}{\mathsf{merge}(x',y')}$$

$$\texttt{(UF7,1)}\quad \frac{\begin{array}{l}\mathsf{merge}(x,y)\\ \mathsf{size}(x,s1)\\ \mathsf{size}(y,s2)\\ s1 \leq s2\end{array}}{\begin{array}{l}\mathrm{del}(\mathsf{merge}(x,y))\\ \mathsf{find.arc}(x,y)\\ \mathrm{del}(\mathsf{size}(y,s2))\\ \mathsf{size}(y,s1+s2)\end{array}} \qquad \texttt{(UF8,1)}\quad \frac{\begin{array}{l}\mathsf{merge}(x,y)\\ \mathsf{size}(x,s1)\\ \mathsf{size}(y,s2)\\ s2 < s1\end{array}}{\begin{array}{l}\mathrm{del}(\mathsf{merge}(x,y))\\ \mathsf{find.arc}(y,x)\\ \mathrm{del}(\mathsf{size}(x,s1))\\ \mathsf{size}(x,s1+s2)\end{array}}$$

Fig. 2. Union-Find

$$\texttt{(ST1,3)}\quad \frac{\begin{array}{l}E(x,c,y)\\ \mathsf{find}(x,z)\\ \mathsf{find}(y,z)\end{array}}{\mathrm{del}(E(x,c,y))} \qquad \texttt{(ST2,c+4)}\quad \frac{E(x,c,y)}{\begin{array}{l}\mathsf{union}(x,y)\\ \mathsf{out}(x,c,y)\end{array}}$$

Fig. 3. Minimum Spanning Tree

are pending, e.g., any state with priority 3 or lower (numerically larger), satisfies the invariant that for any x there is at most one y satisfying $\mathsf{find}(x, y)$ and that the find map defined by the find relation implements the equivalence relation defined by the input union assertions. This implementation uses greedy path compression. The number of visible prefix firings (the abstract running time for fixed priority rules) is $O(n \log n)$ where n is the number of union operations. Figure 3, also taken from [2], shows an $O(n \log n)$ LDP implementation of a minimum spanning tree algorithm based on this implementation of union-find. The input is a set of assertion of the form $E(x, c, y)$ and the output spanning tree is a set of assertions of the form $\mathsf{out}(x, c, y)$. An $O(n\alpha(n))$ implementation of union-find can also be given in LDP but requires that union assertions carry time stamps and that a time-stamped find request is made each time one accesses the find relation. This makes it difficult to write rules, such as rule ST1 in figure 3, whose antecedents notice changes in the find map. In many applications of union-find, such as the spanning tree algorithm in figure 3, the union-find processing time is dominated by other computations (such as the priority queue implicit in rule ST2) and the $O(n \log n)$ implementation of union-find suffices.

5 Polymorphic Type Inference

Polymorphic type inference can be formulated for the lambda calculus extended with an explicit let constructor and a constant for the numeral zero. The lambda calculus with let and zero is the set of expressions defined by the following grammar.

$$x \equiv \mathsf{termvar}(\mathsf{id})$$
$$e \equiv x \mid 0 \mid \mathsf{lambda}(x,\ e) \mid \mathsf{let}(x,\ e_1,\ e_2) \mid \mathsf{apply}(e_1,\ e_2)$$

Note that this grammar represents each expression of the lambda calculus explicitly as an LDP data structure. Variables are represented by LDP data structures of the form termvar(id) where id is the "identifier" of the variable and can be any LDP data structure (ground term). It is interesting to note that one can easily write LDP programs that "gensym" new variables to avoid variable capture in operations such as substitution. More specifically, in implementing substitution on the lambda calculus one must be able to construct a variable x that is guaranteed not to occur in a given expression e. But note that the variable $\mathsf{termvar}(e)$ cannot occur in e. I will write $(e_1\ e_2)$ as an abbreviation for $\mathsf{apply}(e_1,\ e_2)$ and write $(fe_1,\ e_2,\ \ldots,\ e_n)$ as an abbreviation for the "Curried application" $(\ldots((fe_1)e_2)\ldots e_n)$. I will also sometimes write $\lambda x.\ e$ as an abbreviation for $\mathsf{lambda}(x,\ e)$. I will also write $\lambda x_1 x_2.\ e$ for $\lambda x_1.\ \lambda x_2.\ e$

The type inference algorithm described here runs in $O(n\alpha(n) + dn)$ time where d is the scheme depth of the input program. To define scheme depth we first say that the position of e_1 in the let expression $\mathsf{let}(x,\ e_1,\ e_2)$ is a scheme formation position. As described below, these scheme formation positions are the position in the program where polymorphic type inference creates a type scheme. The scheme depth of a position ρ in the program is the number of scheme formation positions above ρ. For example, one might have a sequence of module definitions where each module can use definitions from the previous module. Conceptually, the sequence of module definitions is analogous to the following pattern of let bindings.

$$\mathsf{let}(m_1,\ M_2,\ \mathsf{let}(m_2,\ M_2,\ \mathsf{let}(m_3,\ M_3\ldots)))$$

Each module M_i might consist of a sequence of procedure definitions analogous to the following let bindings.

$$M_i = \mathsf{let}(f_1,\ F_1,\ \mathsf{let}(f_2,\ F_2,\ \mathsf{let}(f_3,\ F_3,\ \ldots)))$$

Each procedure might be defined with a sequence of let expressions as follows

$$f_i = \lambda x_1, \ldots x_n\ \mathsf{let}(y_1,\ e_1,\ \mathsf{let}(y_2,\ e_2,\ \mathsf{let}(y_3,\ e_3,\ \ldots)))$$

Typically the expressions e_i that give the values of let-bound variables inside individual procedure definitions do not themselves contain let expressions. So an arbitrarily long sequence of dependent models, each consisting of an arbitrarily long sequence of dependent procedure definitions, each consisting of an arbitrarily sequence of dependent let expressions, only has scheme depth 3. It seems unlikely that the scheme depth ever exceeds 5 in human written programs.

Hindley-Milner type inference is a process by which the various variables and subexpressions of a given input expression are assigned types. Here we consider only one primitive data type — the type of integers. Finite (nonrecursive) types are generated by the following grammar.

$$\tau \equiv \alpha\ \mid\ \mathsf{int}\ \mid\ \mathsf{arrow}(\tau_1,\ \tau_2)$$

In this grammar α ranges over type variables. The type $\mathsf{arrow}(\tau_1,\ \tau_2)$ is the type assigned to procedures which take as an argument a value in the type τ_1 and produce as an output a value in the type τ_2. I will often write $\tau_1 \rightarrow \tau_2$ as an alternate notation for $\mathsf{arrow}(\tau_1,\ \tau_2)$ and write $\tau_1 \times \cdots \times \tau_n \rightarrow \sigma$ as an alternate notation for $\tau_1 \rightarrow (\tau_2 \rightarrow \cdots \sigma)$. If τ is the type $\sigma \rightarrow \gamma$ then I will sometimes write $\mathsf{domain}(\tau)$ for the type σ and $\mathsf{range}(\tau)$ for the type γ.

The basic constraint in type inference is that in each application $(e_1,\ e_2)$ the expression e_1 must be assigned a type of the form $\tau \rightarrow \sigma$ where e_2 is assigned the type τ and the application $(e_1,\ e_2)$ is assigned the type σ. Type inference is done by unification of type expressions. For each application $(e_1\ e_2)$ we introduce the following constraints.

$$\begin{aligned}
\mathsf{type.of}(e_1,\ \rho_1) &= \mathsf{domain}(\mathsf{type.of}(e_1,\ \rho_1)) \rightarrow \mathsf{range}(\mathsf{type.of}(e_1,\ \rho_1)) \\
\mathsf{type.of}(e_2,\ \rho_2) &= \mathsf{domain}(\mathsf{type.of}(e_1,\ \rho_1)) \\
\mathsf{type.of}((e_1\ e_2),\ \rho_3) &= \mathsf{range}(\mathsf{type.of}(e_1,\ \rho_1))
\end{aligned}$$

Expressions of the form $\mathsf{type.of}(e,\ \rho)$ are treated as type variables. In polymorphic type inference different occurrences of the same expression can be assigned different types as will be explained below. In a type variable of the form

$\mathsf{type.of}(e,\ \rho)$ we have that ρ names a position in the input expression at which the expression e occurs. Unification is used to construct the most general solution to the type constraints.

Figure 4 gives the top level structure of the polymorphic type inference procedure. The procedure definitions are abbreviations for inference rules given in figure 5. The input to the top level procedures in figure 4 is a single expression of the form $\mathsf{do}(\mathsf{analyze}(e,\ \mathsf{root}))$ where e is a closed lambda expression whose variables have been renamed so that each variable is bound at only one position in e. The basic constraints for application are installed by the first procedure in figure 4. The input expression e is typable by polymorphic recursive types if and only if the rules in figures 4, 6, 7, 8, and 9 do not generate clash. These figures give a complete machine-executable implementation of the polymorphic type inference procedure. This procedure allows recursive types by not performing any occurs-check in the unification procedure. Recursive types can be viewed as infinite type expressions with a finite number of distinct subexpressions. For example there is a recursive type τ satisfying the following fixed point equation.

$$\tau = \mathsf{int} \rightarrow \tau$$

It is possible to add occurs-check rules if one wishes to avoid recursive types.

In Hindley-Milner type inference, expressions can have polymorphic types — different occurrences of a procedure can have different types. For example, consider a composition operator comp satisfying the following equation.

$$((\mathsf{comp}\ f\ g)\ x) = (f\ (g\ x))$$

For any types τ_1 and τ_2 the composition operator can be assigned the type $(\tau_1 \rightarrow \tau_2) \times (\tau_1 \rightarrow \tau_2) \rightarrow (\tau_1 \rightarrow \tau_2)$. For the polymorphic composition operator the types τ_1 and τ_2 can be different for different uses of the operator. For example, the same composition operator can be used to compose functions from integers to integers in one place and to compose functions on floating points in another place (in a language with floating point numbers). The type of the polymorphic composition operator is often written as follows.

$$\mathsf{comp} \;:\; \forall \alpha,\ \beta,\ \ (\alpha \rightarrow \beta) \times (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)$$

In Hindley-Milner type inference polymorphism is restricted to let-bound variables. If the top level expression does not involve let then the rules in figures 4 and 6 suffice for type inference. Figures 7, 8, and 9 handle polymorphism. The basic idea is that when the procedure analyzes a let expression it makes the type of the let-bound variable polymorphic. This is done by converting the type to a type scheme which is then instantiated with fresh type variables for each use of the let bound variable.

It is instructive to first consider the case of expressions without let. In this case we can ignore rule A2 and the last procedure definition in figure 4. Since the rules for polymorphic type inference do not involve variable-priority rules, to analyze the run time of the procedure it suffices to count the number of visible

```
procedure analyze(apply(e1, e2), ρ)
  [analyze(e1, down(ρ, 1));
   analyze(e2, down(ρ, 2));
   assert.declare.arrow(type.of(e1, down(ρ, 1)));
   assert.eq.types(type.of(e2, down(ρ, 2)),
                   domain(type.of(e1, down(ρ, 1))));
   assert.eq.types(type.of(apply(e1, e2), ρ),
                   range(type.of(e1, down(ρ, 1))))]

procedure analyze(0, ρ)
  [assert.int.type(type.of(0, ρ))]

        do(analyze(termvar(id), ρ))
        monomorphic(termvar(id))
(A1, 4) ────────
        eq.types(type.of(termvar(id), ρ), type.of(termvar(id)))
        done(analyze(termvar(id), ρ))


        do(analyze(termvar(id), ρ))
        polymorphic(termvar(id), s)
(A2, 4) ────────
        make.copy(s, ρ)
        eq.types(type.of(termvar(id), ρ), , copy(s, ρ))
        done(analyze(termvar(id), ρ))


procedure analyze(lambda(x, e), ρ)
  [assert.monomorphic(x);
   analyze(e, down(ρ, 2));
   assert.declare.arrow(type.of(lambda(x, e), ρ));
   assert.eq.types(type.of(x), domain(type.of(lambda(x, e), ρ)));
   assert.eq.types(type.of(e, down(ρ, 2)), range(type.of(lambda(x, e), ρ))) ]

procedure analyze(let(x, e1, e2), ρ)
  [analyze(e1, down(ρ, 2));
   assert.make.scheme(type.of(e1, down(ρ, 2)), ρ);
   assert.polymorphic(x, sch(type.of(e1, down(ρ, 2)), ρ));
   analyze(e2, down(ρ, 3));
   eq.types(type.of(let(x, e1, e2), ρ), type.of(e2, down(ρ, 3))) ]
```

Fig. 4. Hindley-Milner Type Inference: Fundamental Structure

$$(\mathtt{P1,4})\ \frac{\mathsf{do}(f(t_1,\ \ldots,\ t_n))}{\mathsf{do}([a_1;\ldots;a_n])} \qquad (\mathtt{P2,4})\ \frac{\begin{array}{l}\mathsf{do}(f(t_1,\ \ldots,\ t_n))\\ \mathsf{done}([a_1;\ldots;a_n])\end{array}}{\mathsf{done}(f(t_1,\ \ldots,\ t_n))}$$

$$(\mathtt{P3,4})\ \frac{\mathsf{do}(\mathsf{sequence}(a1,\ a_2))}{\mathsf{do}(\mathsf{a}_1)} \qquad (\mathtt{P4,4})\ \frac{\begin{array}{l}\mathsf{do}(\mathsf{sequence}(a_1,\ a_2))\\ \mathsf{done}(a_1)\end{array}}{\mathsf{do}(a_2)}$$

$$(\mathtt{P5,4})\ \frac{\begin{array}{l}\mathsf{do}(\mathsf{sequence}(a_1,\ a_2))\\ \mathsf{done}(a_1)\\ \mathsf{done}(a_2)\end{array}}{\mathsf{done}(\mathsf{sequence}(a_1,\ a_2))} \qquad (\mathtt{P6\ 1})\ \frac{\mathsf{do}(\mathsf{assert.P}(x_1,\ \ldots,\ x_n))}{\begin{array}{l}P(x_1,\ \ldots,\ x_n)\\ \mathsf{done}(\mathsf{assert.P}(x_1,\ \ldots,\ x_n))\end{array}}$$

Fig. 5. Implementing Procedures. Defining $f(t_1,\ \ldots,\ t_n)$ to be $[a_1;\ldots;a_n]$ abbreviates the rules P1 and P2 above. The notation $[a_1;a_2]$ abbreviates $\mathsf{sequence}(a_1,\ a_2)$ and $[a_1;a_2;\ldots;a_n]$ abbreviates $[a_1;[a_2;\ldots;a_n]]$. Rule P6 is assumed to be present for every action constructor of the form assert.P.

$$(\mathtt{U1,1})\ \frac{\mathsf{eq.types}(\tau,\ \sigma)}{\begin{array}{l}\mathsf{type.var}(\tau)\\ \mathsf{type.var}(\sigma)\\ \mathsf{union}(\tau,\ \sigma)\end{array}} \qquad (\mathtt{U2,\ 1})\ \frac{\mathsf{declare.arrow}(\tau)}{\begin{array}{l}\mathsf{type.var}(\tau)\\ \mathsf{type.var}(\mathsf{domain}(\tau))\\ \mathsf{type.var}(\mathsf{range}(\tau))\\ \mathsf{decl.struct}(\tau,\ \mathsf{domain}(\tau),\ \mathsf{range}(\tau))\end{array}}$$

$$(\mathtt{U3,\ 1})\ \frac{\mathsf{int.type}(\tau)}{\mathsf{type.var}(\tau)} \qquad (\mathtt{U4,1})\ \frac{\mathsf{type.var}(\alpha)}{\mathsf{no.struct}(\alpha)} \qquad (\mathtt{U5,1})\ \frac{\mathsf{struct}(\alpha,\ \beta,\ \gamma)}{\mathsf{del}(\mathsf{no.struct}(\alpha))}$$

$$(\mathtt{U6,2})\ \frac{\begin{array}{l}\mathsf{decl.struct}(\alpha,\ \beta,\ \gamma)\\ \mathsf{no.struct}(\alpha)\end{array}}{\mathsf{struct}(\alpha,\ \beta,\ \gamma)} \qquad (\mathtt{U7,1})\ \frac{\begin{array}{l}\mathsf{decl.struct}(\alpha,\ \beta,\ \gamma)\\ \mathsf{struct}(\alpha,\ \beta',\ \gamma')\end{array}}{\begin{array}{l}\mathsf{union}(\beta,\ \beta')\\ \mathsf{union}(\gamma,\ \gamma')\end{array}}$$

$$(\mathtt{U8,1})\ \frac{\begin{array}{l}\mathsf{struct}(\alpha,\ \beta,\ \gamma)\\ \mathsf{find.arc}(\alpha,\ \alpha')\end{array}}{\mathsf{decl.struct}(\alpha',\ \beta,\ \gamma)} \qquad (\mathtt{U9,1})\ \frac{\begin{array}{l}\mathsf{int.type}(\tau)\\ \mathsf{find.arc}(\tau,\ \sigma)\end{array}}{\mathsf{int.type}(\sigma)} \qquad (\mathtt{U10,1})\ \frac{\begin{array}{l}\mathsf{struct}(\alpha,\ \beta,\ \gamma)\\ \mathsf{int.type}(\alpha)\end{array}}{\mathsf{clash}}$$

Fig. 6. Type Unification

prefix firings. The number of prefix firings of the rules implicit in figure 4 is proportional to the number of expressions analyzed, i.e., the number of positions occupied in the input expression. The number of occupied positions corresponds to the written length of the expression and I will simple call this number "the size of the input" and denote it by n. To finish the analysis it suffices to count the number of visible prefix firings in the union-find rules in figure 2 and of the unification rules in figure 6. The input to the unification module consists of assertions of the form $\mathsf{declare.arrow}(\sigma)$, $\mathsf{int.type}(\sigma)$ and $\mathsf{eq.types}(\sigma,\ \tau)$. The output is the assertion clash if the constraints are unsatisfiable and otherwise the output consists of the union-find equivalence relation on the set of type expressions plus assertions of the form $\mathsf{struct}(\alpha,\ \beta,\ \gamma)$ which implies that $\alpha = \beta \rightarrow \gamma$. The rules maintain the invariant that for any α there is at most one assertion of the form $\mathsf{struct}(\alpha,\ \beta,\ \gamma)$. Note that, with the exception of rule U2, the unification rules do not introduce new expressions. In fact the number of expressions generated by the rules is no more than linear in the number of assertions input to the unification process. This implies that the number of (uncompressed) underlying find arcs in the union-find assertions is also no more than linear in the number of inputs to unification. The number of prefix firings of the unification rules can be seen to be proportional to the number of find arcs and hence the number of prefix firings of the unification rules is proportional to the number of input assertions. In let-free programs the number of inputs to unification is proportional to the number of positions occupied in the input expression. Because the unification rules use only the uncompressed find arcs a lazy path compression unification algorithm can be used. This gives $n\alpha(n)$ type inference procedure for let-free programs. The lazy path compression algorithm can also be used with the full polymorphic type inference procedure. While linear time unification is appropriate for let-free programs, in the presence of polymorphism one must alternate unification processing with type scheme construction and this alternation prevents the use of linear time unification.

To understand the subtleties of the polymorphic case consider the following expression.

$$\lambda y.\mathsf{let}(f,\ \lambda x.y,\ e)$$

In this case multiple occurrences of f in e can have different domain types but must all have the same range type. The range type of f is the type of the lambda-bound variable y which can be determined from the contexts into which the overall lambda expression is embedded. In the Hindley-Milner type inference algorithm we assign f and y the following type.

$$\begin{aligned} y &: \beta \\ f &: \forall \alpha,\ \alpha \rightarrow \beta \end{aligned}$$

Note that in this type α is a bound type variable while β is a free type variable. In general, when the procedure constructs a type scheme it must distinguish type variables which can be universally quantified (the bound type variables) from the type variables which must be left free. Each type variable has a position in the program where it is created. In a let expression $\mathsf{let}(x,\ e_1,\ e_2)$ type variables

$$\texttt{(MS1,1)}\ \frac{\mathsf{type.var}(\alpha)}{\mathsf{unbound}(\alpha)} \qquad \texttt{(MS2,1)}\ \frac{\mathsf{arrow.type}(\alpha)}{\mathrm{del}(\mathsf{unbound}(\alpha))} \qquad \texttt{(MS3,1)}\ \frac{\mathsf{int.type}(\alpha)}{\mathrm{del}(\mathsf{unbound}(\alpha))}$$

$$\texttt{(MS4,1)}\ \frac{\begin{array}{l}\mathsf{make.scheme}(\tau,\ \rho)\\ \mathsf{find}(\tau,\ \tau')\\ \mathsf{struct}(\tau',\ \sigma,\ \gamma)\end{array}}{\begin{array}{l}\mathsf{sch.struct}(\mathsf{sch}(\tau,\ \rho),\ \mathsf{sch}(\sigma,\ \rho),\ \mathsf{sch}(\gamma, \rho))\\ \mathsf{make.scheme}(\sigma,\ \rho)\\ \mathsf{make.scheme}(\gamma,\ \rho)\end{array}} \qquad \texttt{(MS5,1)}\ \frac{\begin{array}{l}\mathsf{make.scheme}(\tau,\ \rho)\\ \mathsf{find}(\tau,\ \tau')\\ \mathsf{int.type}(\tau')\end{array}}{\mathsf{int.scheme}(\mathsf{sch}(\tau,\ \rho))}$$

$$\texttt{(MS6,2)}\ \frac{\begin{array}{l}\mathsf{make.scheme}(\tau,\ \rho)\\ \mathsf{find}(\tau,\ \tau')\\ \mathsf{unbound}(\tau')\\ \mathsf{scheme.depth}(\tau',\ i)\\ \mathsf{scheme.depth}(\rho,\ j)\\ i < j\end{array}}{\mathsf{free.var}(\mathsf{sch}(\tau,\ \rho),\ \tau')} \qquad \texttt{(MS7,2)}\ \frac{\begin{array}{l}\mathsf{make.scheme}(\tau,\ \rho)\\ \mathsf{find}(\tau,\ \tau')\\ \mathsf{unbound}(\tau')\\ \mathsf{scheme.depth}(\tau',\ i)\\ \mathsf{scheme.depth}(\rho,\ j)\\ j \leq i\end{array}}{\mathsf{bound.var}(\mathsf{sch}(\tau,\ \rho),\ \mathsf{sch}(\tau',\ \rho))}$$

Fig. 7. Making a Type Scheme

$$\texttt{(C1,3)}\ \frac{\begin{array}{l}\mathsf{make.copy}(s,\ \rho)\\ \mathsf{sch.struct}(s,\ t,\ v)\end{array}}{\begin{array}{l}\mathsf{declare.arrow}(\mathsf{copy}(s,\ \rho))\\ \mathsf{eq.types}(\mathsf{domain}(\mathsf{copy}(s,\ \rho)),\ \mathsf{copy}(t,\ \rho))\\ \mathsf{eq.types}(\mathsf{range}(\mathsf{copy}(s,\ \rho)),\ \mathsf{copy}(v,\ \rho))\\ \mathsf{make.copy}(t,\ \rho)\\ \mathsf{make.copy}(v,\ \rho)\end{array}} \qquad \texttt{(C2,3)}\ \frac{\begin{array}{l}\mathsf{make.copy}(s,\ \rho)\\ \mathsf{int.scheme}(s)\end{array}}{\mathsf{int.type}(\mathsf{copy}(s, \rho))}$$

$$\texttt{(C3,3)}\ \frac{\begin{array}{l}\mathsf{make.copy}(s,\ \rho)\\ \mathsf{free.var}(s,\ \tau')\end{array}}{\mathsf{eq.types}(\mathsf{copy}(s, \rho),\ \tau')} \qquad \texttt{(C4,3)}\ \frac{\begin{array}{l}\mathsf{make.copy}(s,\ \rho)\\ \mathsf{bound.var}(s_1,\ s_2)\end{array}}{\mathsf{eq.types}(\mathsf{copy}(s_1, \rho),\ \mathsf{copy}(s_2,\ \rho))}$$

Fig. 8. Copying a Type Scheme at Program Position ρ

$$\texttt{(D1,1)}\ \frac{}{\mathsf{scheme.depth}(\mathsf{root},\ 0)}$$

$$\texttt{(D2,1)}\ \frac{\begin{array}{l}\mathsf{do}(\mathsf{analyze}(\mathsf{let}(x,\ e_1,\ e_2),\ \rho))\\ \mathsf{scheme.depth}(\rho,\ d)\end{array}}{\begin{array}{l}\mathsf{scheme.depth}(\mathsf{down}(\rho,\ 2),\ d+1)\\ \mathsf{scheme.depth}(\mathsf{down}(\rho,\ 3),\ d)\end{array}}$$

$$\texttt{(D3,1)}\ \frac{\begin{array}{l}\mathsf{do}(\mathsf{analyze}(\mathsf{lambda}(x,\ e),\ \rho))\\ \mathsf{scheme.depth}(\rho,\ d)\end{array}}{\mathsf{scheme.depth}(\mathsf{down}(\rho,\ 2),\ d)}$$

$$\texttt{(D4,1)}\ \frac{\begin{array}{l}\mathsf{do}(\mathsf{analyze}(\mathsf{apply}(e_1,\ e_2),\ \rho))\\ \mathsf{scheme.depth}(\rho,\ d)\end{array}}{\begin{array}{l}\mathsf{scheme.depth}(\mathsf{down}(\rho,\ 1),\ d)\\ \mathsf{scheme.depth}(\mathsf{down}(\rho,\ 2),\ d)\end{array}}$$

$$\texttt{(D5,1)}\ \frac{\begin{array}{l}\mathsf{type.var}(\mathsf{type.of}(e,\ \rho))\\ \mathsf{scheme.depth}(\rho,\ d)\end{array}}{\mathsf{scheme.depth}(\mathsf{type.of}(e,\ \rho),\ d)}$$

$$\texttt{(D6,1)}\ \frac{\begin{array}{l}\mathsf{type.var}(\mathsf{copy}(s,\ \rho))\\ \mathsf{scheme.depth}(\rho,\ d)\end{array}}{\mathsf{scheme.depth}(\mathsf{copy}(s,\ \rho),\ d)}$$

$$\texttt{(D7,1)}\ \frac{\begin{array}{l}\mathsf{scheme.depth}(x,\ d_1)\\ \mathsf{scheme.depth}(x,\ d_2)\\ d_1 < d_2\end{array}}{\mathrm{del}(\mathsf{scheme.depth}(x,\ d_2))}$$

$$\texttt{(D8,2)}\ \frac{\begin{array}{l}\mathsf{scheme.depth}(x,\ d)\\ \mathsf{find.arc}(x,\ y)\end{array}}{\mathsf{scheme.depth}(y,\ d)}$$

$$\texttt{(D9,3)}\ \frac{\begin{array}{l}\mathsf{scheme.depth}(\alpha,\ d)\\ \mathsf{struct}(\alpha,\ \beta,\ \gamma)\end{array}}{\begin{array}{l}\mathsf{scheme.depth}(\beta,\ d)\\ \mathsf{scheme.depth}(\gamma,\ d)\end{array}}$$

Fig. 9. The Computation of Scheme Depth

created in e_1 are candidates for quantification in the type scheme. However, the expression e_1 usually contains free variables such as y in the above example. The types associated with the free variables in e_1 cannot be generalized. A type that is provably equal to a range type or a domain type of an ungeneralizable type is also ungeneralizable. To determine which types are generalizable we us the notion of scheme depth defined earlier. The formal definition of the scheme depth of a position is given by rules D1 through D4 in figure 9. We first say that a type α is provably a subexpression of a type β if α is provably equal to β (the union-find structure equates them) or α is of the form $\mathsf{domain}(\gamma)$ or $\mathsf{range}(\gamma)$ where γ is provably a subexpression of β. Each type variable has a creation depth and an inferred depth. The inferred depth of α is the creation depth of the shallowest type variable β such that α is provably a subexpression of β. Rules D5 and D6 in figure 9 install the creation depth of type variables and rules D7, D8, and D9 compute inferred depth. A variable can be bound by the universal quantifier at scheme creation time as long as its inferred depth is no less than the depth of position at which the scheme is being created.

Figure 7 gives rules for creating type schemes. We leave it to reader to decipher the meaning of these rules. One point concerning their complexity should be noted however. The find assertions in the antecedents of these rules obviously should only be instantiation with the find values that are in place when the scheme is constructed. This can be achieved by adding an antecedent of the form $\mathsf{active}(\rho)$ to each rule where this assertion is only true during the computation of the scheme for position ρ. With this modification it is easy to check that the number of visible prefix firings of the rules in figure 7 is proportional to the number of nodes in the scheme created. For programs of bounded order and arity there is a constant upper bound on the number of nodes in each scheme. The rules for making a scheme are consistent with the use a lazy path compression version of union-find.

The rules in figure 8 copy a type scheme at a particular program location. The number of visible prefix firings of these rules is proportional to the size of the scheme being copied. If schemes have a bounded size then the total time taken to create and copy schemes is proportional to the size of the input program (the number of occupied positions). In this case the total number of type variables is also proportional the size of the input program so the time spent inside a lazy path compression version of union-find is $O(n\alpha(n))$. The number of visible prefix firings of rule D7, D8, and D9 in bounded by the size of the program (the number of type variables) times the number of depths, i.e., the scheme depth of the deepest scheme position. This gives a total running time of $O(n\alpha(n) + dn)$ where n is the length of the program and d is the scheme depth of the program.

6 Conclusions

This paper is a case study in the use of bottom-up logic programming extended with priorities and deletions in presenting and analyzing algorithms. Efficient Hindley-Miler type inference seems to be a challenging algorithm to both formulate and analyze. The algorithm presented here is essentially the one used in ML and Caml implementations. Although beauty is in the eye of the beholder, it can at least be argued that the inference rule presentation of the algorithm is clearer both respect to its correctness *and with respect to its running time* than is any presentation of the algorithm using traditional control structures. Of course the top level procedure in figure 4 uses classical procedural control structures. However, the implementation of unification and the depth propagation seems easier in inference rules that classical control structures. Also, inference rules provide a different form of modularity than is possible in classical programs. The unification algorithm can be given as a separate module that uses a union-find module without modification. Depth propagation can be done with yet another module. The complexity analysis of these modules can be done separately (modularly). This modularization seems difficult with classical control structures. The use of inference rules also seems to facilitates the treatment of recursive types.

In comparing the algorithm in figures 4, 6, 7, 8, and 9 to other presentations it is important to keep in mind that the these figures contain machine readable code. Machine readable code is usually less clear than informal descriptions of algorithms written in English. A fair comparison would require comparing these figures to a machine-readable implementation in an some more traditional programming language.

Bottom-up logic programming is clearly not the best tool for all applications. However, it is hoped that this case study demonstrates that presenting and analyzing complex algorithms as bottom-up logic programs is indeed feasible and perhaps provides greater clarity than traditional approaches.

References

1. H. Ganzinger and D. McAllester. A new meta-complexity theorem for bottom-up logic programs. In *Proc. International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*, pages 514–528. Springer-Verlag, 2001.
2. H. Ganzinger and D. McAllester. Logical algorithms. In *Proc. International Conference on Logic Programming (ICLP)*, 2002.
3. F. Henglein. Type inference with polymorphic recursion. *Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):253–289, 1993.
4. A. Kfoury, J. Tiuryn, and P. Urzyczyn. Ml typability is dexptime complete. In *Proceedings of the 15th Colloquium on Trees in Algebra and Programming (CAAP)*, pages 468–476. ACM, 1990.
5. H. Mairson. Deciding ml typability is complete for deterministic exponential time. In *ACM Symposium on Principles of Programming Languages*. Association for Computing Machinery, 1990.
6. David McAllester. On the complexity analysis of static analyses. *Journal of the ACM (JACM)*, 49(4):512–537, 2002. A short version appeared in SAS99.
7. Robin Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978.
8. Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992.