Let-Polymorphism and Eager Type Schemes

Chuck Liang

University of Pennsylvania
200 S. 33rd Street
Philadelphia, PA 19104-6389
email: liang@saul.cis.upenn.edu

Abstract

This paper presents a new algorithm for polymorphic type inferencing involving the `let` contruct of ML. It avoids the *gen* operation of the algorithm W of Damas and Milner. This new algorithm is a closer counterpart to the proof-theoretic formulation of the ML typing discipline than other approaches. Sketches of the proofs of correctness, including completeness for the `let` case, are also given. The basic technique of the algorithm also facilitates the declarative formulation of type inference as goal-directed proof-search.

Introduction

Various formulations and algorithms for the inference or assignment of principal types to untyped λ-terms have long existed. In [2] Damas and Milner extended type-inferencing to involve the polymorphic `let` construct of functional programming langugaes (ML). They formulated a declarative, proof-theoretic calculus for the ML type system, given here in Figure 1. Unfortunately, this calculus does not by itself lead directly to an inference algorithm that yields principal type schemes. For this purpose the algorithm "W" was given. Algorithm W involves an operation called *gen* (or *close*) in typing `let`-expressions. Together with the unification algorithm that yields most-general unifiers, this operation ensures maximal generality of the type scheme for the locally-bound term in `let` expressions. With respect to the original Damas-Milner calculus, *gen* effectively represents a *forward-chaining* step. The consequence of using this operation is that a simple proof of completeness of the algorithm with respect to the calculus can not be given. Furthermore, it is also desirable to formulate principal type inference as deterministic, goal-directed proof search (augmented with unification) in some proof system. The *gen* operation has proved to be a main obstacle in formulating such a system satisfactorily.

$$\textbf{Proj} \qquad \frac{}{H \vdash x : T, \quad x : T \in H}$$

$$\textbf{abs} \qquad \frac{H, x : s \vdash M : t}{H \vdash \lambda x.M : s \rightarrow t}$$

$$\textbf{app} \qquad \frac{H \vdash M : s \rightarrow t \quad H \vdash N : s}{H \vdash (M\ N) : t}$$

$$\textbf{let} \qquad \frac{H \vdash M : S \quad H, x : S \vdash N : t}{H \vdash \texttt{let}\ x = M\ \texttt{in}\ N\ :\ t}$$

$$\Pi\textbf{-Intro} \qquad \frac{H \vdash M : T \quad a \text{ not free in } H}{H \vdash M : \Pi a.T}$$

$$\Pi\textbf{-Elim} \qquad \frac{H \vdash M : \Pi a.T}{H \vdash M : T[s/a]}$$

(s, t represent unquantified types; S, T represent arbitrary type schemes)

Figure 1: The Damas-Milner Calculus [2]

In this paper we present an alternative algorithm for type inference that avoids the use of gen[1]. This algorithm also takes advantage of the fact that in practice, only closed type environments are needed. With closed environments, all free type variables that are dynamically introduced during the type inferencing process can be safely discharged (Π-quantified) upon successful completion of the process. Only in the inductive proofs of correctness need we be concerned with the more general case of open environments.

This paper is organized as follows. In Section 1 we motivate and present our algorithm. In Section 2 we give some sample type inferences using the algorithm. Sketches of correctness proofs are given in Section 3. In Section 4 we discuss applications of our technique with respect to separate compilation and other issues, including the formulation of type inference as proof search. We will also discuss related work, in particular those of Leivant [9], Appel and Shao [1], and Harper [5].

1 Of Variables, Free, Bound, and Fugitive

Strictly speaking, the algorithm W infers *types,* and not *type schemes.* Let $\overline{v_m}$ denote $v_1, \ldots, v_m$. Whenever a typing assumption $f : \Pi\overline{v_m}.t$ is used, a "copy" of the type $t[\overline{x_m}/\overline{v_m}]$ is created using a set of new free variables $\overline{x_m}$. This occurs uniformly except in the `let` case, when *type scheme* inference takes place in the form of applying gen. The technique we use approaches type inference from the opposite direction. Here *type scheme* inference is the default. In other words, we shall always try to keep type variables Π-quantified as much as possible. If the typing of a compound expression e requires two instances of a type scheme $\Pi\overline{v}.t$, this is made possible by *appending* two

[1] and which does not merely replace $let\ x = M\ in\ N$ with $N[M/x]$; this replacement leads to redundant inferences.

copies of the quantifier prefix to yield $\Pi\overline{v_m}\Pi\overline{v'_m}.s$, where s is the type of e. New free variables are uniformly replaced by new Π-bound variables. Typing conflicts are resolved *post-hoc* to prevent over-generalization.

Before further discussion, we first present the algorithm. Define an *extended type environment* H_e to be a mapping from program (or term) variables x to a structure $\Lambda\overline{v_m}.(\sigma, t)$, which we shall call an *eager type scheme*. Here, σ is a substitution on type variables and t is a type such that $\sigma(t) = t$. The binding construct Λ quantifies over the type variables $\overline{v_m}$, which may occur anywhere in the substitution-type pair (σ, t). The intuitive meaning of this mapping is that x maps to the *potential* type scheme $\Pi\overline{v_m}.t$ if the substitution σ is applied to the current type environment. The algorithm, which we shall call W_Π, is given in Figure 2.

For an extended type environment H_e and a program expression M, $W_\Pi(H_e;\ M)$ returns a structure $\Lambda\overline{v_m}.(\sigma, t)$. Let $\emptyset$ represent the empty (or identity) substitution. Assume all substitutions are idempotent ($\theta \circ \theta = \theta$). The operation *join* is basically the same as the *join* operation from Leivant [9][2] . Given substitutions $S_1, \ldots, S_n$, $join(S_1, \ldots, S_n) = R$ such that for each S_i in $S_1, \ldots, S_n$ there is a substitution P_i such that $P_i \circ S_i = R$. Furthermore, if R' also satisfies this property then there is a substitution P such that $P \circ R = R'$. That is, $join(S_1, \ldots, S_n)$ is the most general common instance of $S_1, \ldots, S_n$ (if it exists). The *join* operation can be implemented using the standard unification algorithm.

The use of α-equivalence ($=_\alpha$) in the definition of the algorithm is appropriate since the Λ binder can be conveniently represented by λ-abstraction in the λ-calculus. This amounts to using *higher-order abstract syntax* [12] to simplify our presentation. We use "Λ" to distinguish it from the "λ" used in program expressions. Higher-order notation is not an essential part of the algorithm (though it was the original impetus for the basic technique involved).

To explain how this algorithm is used relative to a regular (non-extended) type environment, we define the following:

Definition 1 *(Base Extension)*
Given a type environment H, let $H\uparrow$ represent the extended type environment that includes $(x \mapsto \Lambda\ \overline{v_m}.(\emptyset,\ t))$ for each $(x : \Pi\overline{v_m}.t)$ in H.

For a **closed** type environment H, if $W_\Pi(H\uparrow; M)$ succeeds with $\Lambda\overline{v_m}.(\sigma, t)$ then it will always be the case that (σ, t) contains no free variables. We can therefore conclude that $H \vdash M : \Pi\overline{v_m}.t$.

The crucial point in W_Π where "free variables" are dynamically introduced into an environment occurs in the typing of a λ-expression $\lambda x.M$. Here x is assumed to have type a, where a is a

[2]The only difference with Leivant's *join* operation is that it also returns a sequence of substitutions $P_1, \ldots, P_n$ as well as the final, resolved common substitution: $join(S_1, \ldots, S_n) = (R,\ P_1, \ldots, P_n)$ such that $P_i \circ S_i = R$.

$\boldsymbol{W_{\Pi}}(\boldsymbol{H}_e;\ \boldsymbol{x}) =$ $H_e(x)$, for program variable x.

$\boldsymbol{W_{\Pi}}(\boldsymbol{H}_e;\ \boldsymbol{\lambda x.M}) =$ let a be a new type variable, and let

$$W_{\Pi}(H_e, x \mapsto (\emptyset, a);\ M) =_{\alpha} \Lambda\overline{v_n}.(\sigma,\ t).$$

Return

$$\Lambda a \Lambda\overline{v_n}.(\sigma,\ \sigma(a \rightarrow t)).$$

$\boldsymbol{W_{\Pi}}(\boldsymbol{H}_e;\ (\boldsymbol{M}\ \boldsymbol{N})) =$ let

$$W_{\Pi}(H_e;\ M) =_{\alpha} \Lambda\overline{v_n}.(\sigma_1,\ t_1), \quad \text{and} \quad W_{\Pi}(H_e;\ N) =_{\alpha} \Lambda\overline{u_m}.(\sigma_2,\ t_2)$$

such that the bound variables $\overline{u_m}$ are distinct from $\overline{v_n}$. For a new type variable t_v, let θ be the most general unifier of t_1 and $t_2 \rightarrow t_v$. Let $\sigma = join(\theta, \sigma_2, \sigma_1)$. Return

$$\Lambda t_v \Lambda\overline{u_m} \Lambda\overline{v_n}.(\sigma,\ \sigma(t_v)).$$

$\boldsymbol{W_{\Pi}}(\boldsymbol{H}_e;\ \boldsymbol{let\ x = M\ in\ N}) =$ let

$$W_{\Pi}(H_e;\ M) =_{\alpha} \Lambda\overline{v_n}.(\sigma_1,\ t_1), \quad \text{and}$$

$$W_{\Pi}(H_e, (x \mapsto \Lambda\overline{v_n}.(\sigma_1,\ t_1));\ N) =_{\alpha} \Lambda\overline{u_m}.(\sigma_2,\ t_2)$$

such that $\overline{u_m}$ are distinct from $\overline{v_n}$. Let $\sigma = join(\sigma_2, \sigma_1)$. Return

$$\Lambda\overline{u_m} \Lambda\overline{v_n}.(\sigma,\ \sigma(t_2)).$$

Figure 2: Algorithm W_{Π}

new type variable. This variable is free only in the dynamic, temporary environment. It will be captured by Λ-abstraction when the top-level type scheme of $\lambda x.M$ is constructed. We will call such variables introduced for λ-bindings *fugitive* variables. In the algorithm W of Damas-Milner, new free variables are constantly being generated. We observe, however, that if the initial environment is closed then all dynamically generated free variables that can *not* be immediately quantified are those that result from unification with fugitive variables. But since the fugitives themselves will also be quantifiable eventually, any new variable that occurs in a substitution for them will also be quantifiable eventually. In algorithm W_Π, instead of using new free variables, we immediately quantify any new variable generated from discharging (an instance of) a typing assumption. As a consequence, some invalid expressions will "appear momentarily typable." The *join* operation, however, will catch any inconsistencies in the substitutions and reject untypable expressions. The technique presented here can be summed up as "eager quantification, delayed resolution." We will illustrate this principle with three examples.

2 Sample Inferences

Assume the type environment H contains the assignment $f : \Pi v.v \to v$. Consider typing the expression $\lambda x.(f\ x)$. First we augment the extended environment $H\uparrow$ with $x \mapsto (\emptyset, a)$ for a new fugitive variable a. In typing $(f\ x)$, we unify $v \to v$ with $a \to t_v$ for some new variable t_v. Thus $W_\Pi(H\uparrow, x \mapsto (\emptyset, a);\ (f\ x)) =_\alpha \Lambda t_v \Lambda v.([v/a, v/t_v], v)$. The accompanying substitution is then applied to $a \to v$, and the fugitive a is captured, yielding $\Lambda a \Lambda t_v \Lambda v.([v/a, v/t_v],\ (v \to v)$. We can therefore conclude that $H \vdash \lambda x.(f\ x) : \Pi a \Pi t_v \Pi v.v \to v$.

Now consider $let\ x = \lambda y.y\ in\ (x\ x)$. First, $\lambda y.y$ is inferred as having the eager type scheme $\Lambda v.(\emptyset,\ v \to v)$. Then x is assumed to map to this eager scheme. For $(x\ x)$, the type of x is inferred twice as $\Lambda v.(\emptyset,\ v \to v)$ and $\Lambda w.(\emptyset,\ w \to w)$. With a new variable t_v, $(w \to w) \to t_v$ is unified with $v \to v$, yielding the substitution $[w \to w/t_v, w \to w/v]$. This substitution can be trivially joined with the two instances of the empty substitution. Thus calling W_Π on $(x\ x)$ will return the structure

$$\Lambda t_v \Lambda w \Lambda v.([w \to w/t_v, w \to w/v],\ w \to w),$$

and since the substitution returned joins immediately with the empty substitution, we can conclude that $let\ x = \lambda y.y\ in\ (x\ x)$ has type $\Pi w.w \to w$ (eliminating the vacuous quantifiers this time for convenience; we may also implement this elimination as an optimization). The key observation here is that a type *scheme* is always inferred, thereby eliminating the need for the *gen* operation.

For the final example, assume the program variable p has type $\Pi v.v \to v \to v$. Consider the *untypable* expression $\lambda y.(let\ x = (p\ y)\ in\ (x\ x))$. For the top level λ-abstraction, a new fugitive variable a is assumed as the type for y. In the `let` expression, $(p\ y)$ can be inferred as having the structure $\Lambda t_v \Lambda v.([v/a, (v \to v)/t_v],\ v \to v)$. The program variable x is then assumed to map to this structure

in the updated extended environment. Typing $(x\ x)$ will again produce two individual copies of this structure:

$$\Lambda t_v.\Lambda v.([v/a,(v\to v)/t_v],\ v\to v),\quad \text{and}\quad \Lambda t_w.\Lambda w.([w/a,(w\to w)/t_w],\ w\to w).$$

Another type variable t_z is introduced, and $(w\to w)\to t_z$ is unified with $v\to v$, resulting in the substitution $[(w\to w)/v,(w\to w)/t_z]$. But this substitution can not be joined with the two substitutions from the individual recursive inferences for y: $[v/a,(v\to v)/t_v]$, and $[w/a,(w\to w)/t_w]$. The variable a can not have both $w\to w$ and w (or both $v\to v$ and v) as instances.

Notice that although a fugitive a is a (dynamically) free variable, it can be substituted by a (Λ) bound variable, as when a was substituted by the Λ-bound variable v in the third example. Once a variable is bound, "copies can be made", and thus two instances of v, v and w, were created. Type inference was allowed to continue where in algorithm W it would have failed: v was unified with $w\to w$. This "eager inference," however, was invalidated when the substitutions were joined, revealing that v/a and w/a are inconsistent if $v = w\to w$. In case these substitutions can be successfully joined, then these variables (v and w) can remain rightfully quantified, since the final type scheme returned will quantify over all fugitive variables. Because we need to keep track of which bound variables are in fact "eagerly" quantified, the *join* operation must replace the composition of substitutions as used in algorithm W. That is, we need to "memorize" the various substitutions for the fugitive variables in the form of extended type environments.

3 Correctness Proofs

This section addresses the major components required to show soundness and in particular completeness of W_Π with respect to principal type schemes for the Damas-Milner typing discipline. As a consequence we also show how to extend the algorithm to accommodate open type environments in general.

With respect to a structure $\Lambda\overline{v_m}.(\sigma,\ t)$, we say that a bound variable v_i is *innocent* if for some free variable (or fugitive) a, $\sigma(a) = t$ such that v_i occurs in t. That is, innocent variables are variables that were Λ-bound prematurely, and should be freed if a occurs in the environment.

Definition 2 *(Base Compression)*
Given an extended type environment H_e of the form

$$\{x_1\mapsto \Lambda\overline{v^1_{n_1}}(\sigma_1,\ t_1),\ldots,x_m\mapsto(\Lambda\overline{v^m_{n_m}}.(\sigma_m,\ t_m)\}.$$

Assume that all variables $v^i_{j_k}$ are distinct. Let $\delta = join(\sigma_1,\ldots,\sigma_m)$. Let $\overline{u_k}$ be all the variables in δ that are innocent. Let $\overline{w_l}$ be all the variables $\overline{v^1_{n_1}},\ldots,\overline{v^m_{n_m}}$ minus $\overline{u_k}$. Define $\boldsymbol{H}_e\downarrow = (\delta, H)$ where

H is the type environment

$$\{x_1 : \Pi\overline{w_l}.\delta(t_1), \ldots, x_m : \Pi\overline{w_l}.\delta(t_m)\}.$$

For a type environment H, clearly $H\uparrow\downarrow = (\emptyset, H)$.

Theorem 1 *Given an extended type environment H_e and a program expression M, assume $W_\Pi(H_e; M) = \Lambda\overline{v_m}.(\sigma, t)$. If $[H_e, y \mapsto \Lambda\overline{v_m}.(\sigma, t)]\downarrow = (\delta, H)$ for some new "dummy" variable y, then $H \vdash M : H(y)$.*

Proof: By structural induction on M, appealing to properties of the *join* operation. □.

We forgo the details of the soundness proof in favor of completeness. The following corollary establishes soundness for closed type environments.

Corollary 1.1 *(Soundness of W_Π)*
Given a closed type environment H and a term M, $W_\Pi(H\uparrow; M) = \Lambda\overline{v_m}.(\sigma, t)$ implies $H \vdash M : \Pi\overline{v_m}.t$.

The (syntactic) completeness proof uses the basic technique of Leivant [9] of restating the inference rules in more desirable forms. The main contribution here is our `let` case. A *generic application* of a substitution G to a type scheme $\Pi\overline{v_m}.t$ is defined as $G[\Pi\overline{v_m}.t] = \Pi\overline{v_m}.G(t)$. That is, generic application can replace bound variables as well as free variables. For every "generic instance" (in the sense of Damas-Milner [2]) σ' of σ there is a substitution G such that $G[\sigma] = \sigma'$ (modulo some vacuous Π quantifiers). Because the $\downarrow$ operation breaks quantifiers, the completeness theorem must be stated using generic applications of substitutions. In the theorem below, we assume that all variables (free and bound) in H_e are distinct.

Theorem 2 *Assume for the extended type environment H_e, $H_e\downarrow$ exists and is equal to (δ, H). Assume $S[H] \vdash M : T$ for substitution S, term M and type scheme T. Then $W_\Pi(H_e; M) = \Lambda\overline{v_m}.(\sigma, t)$. For a new term variable y, let $[H_e, y \mapsto \Lambda\overline{v_m}.(\sigma, t)]\downarrow = (\delta', H')$, let $\theta \circ \delta = \delta'$,[3] and let $H'(y) = \Pi\overline{w_l}.t'$. It also holds that there exists a substitution ρ such that $\rho \circ \theta = S$ and $\rho[\Pi\overline{w_l}.t'] = T$.*

Proof: By induction on the height of derivations. For the inductive basis if $x : \Pi\overline{w_l}.t_0 \in H$ then $x \mapsto \Lambda\overline{v_m}.(\sigma,\ t) \in H_e$ for some σ and t, and $W_\Pi(H_e; x) = \Lambda\overline{v_m}.(\sigma,\ t)$. Here, $\delta' = \delta$. We set $\rho = S$ in this case and the result follows. The Π-**Elim** case is trivial. The Π-**Intro** case also follows easily since all variables not free in H_e are always Λ-bound. The **abs** and **app** cases can be shown by

[3] We know θ exists since $\delta' = join(\delta, \sigma)$.

adapting the following technique found in Leivant [9]:[4] the inference rules can be rewritten in the forms

$$\frac{S[H], x : S[a] \vdash M : S[b]}{S[H] \vdash \lambda x.M : S[a \rightarrow b]}\ abs \qquad \frac{S[H] \vdash M : S[r \rightarrow t_v] \quad S[H] \vdash N : S[r]}{S[H] \vdash (M\ N) : S[t_v]}\ app,$$

where a, b, r and t_v are distinct type variables not appearing elsewhere.

We concentrate on the **let** case. Let $H_e \downarrow = (\delta, H)$. A **let** rule-application can be written in the form

$$\frac{S[H] \vdash M : \xi \quad S[H], x : \xi \vdash N : T}{S[H] \vdash \texttt{let}\ x = M\ \texttt{in}\ N\ :\ T}\ let,$$

where ξ is some type *scheme*. By inductive hypothesis, $W_\Pi(H_e; M) = \Lambda\overline{v_m}.(\sigma_1, t_1)$ such that $[H_e, y \mapsto \Lambda\overline{v_m}.(\sigma_1, t_1)] \downarrow = (\delta_1, H_1)$. Let $H_1(y) = \Pi\overline{w_l}.t$ and $\theta \circ \delta = \delta_1$. There is also a substitution ρ_1 such that $\rho_1 \circ \theta = S$ and $\rho_1[\Pi\overline{w_l}.t] = \xi$. But $\rho_1(t) = \rho_1(\delta_1(t_1))$ by definition of H_1, and $\rho_1(\delta_1(t_1)) = \rho_1(\theta(\theta(\delta(t_1)))) = S(\delta_1(t_1))$. Thus $\xi = S[\Pi\overline{w_l}.\delta_1(t_1)]$. We can therefore rewrite the above instance of the **let** rule as:

$$\frac{S[H] \vdash M : S[\Pi\overline{w_l}.\delta_1(t_1)] \quad S[H, x : \Pi\overline{w_l}.\delta_1(t_1)] \vdash N : T}{S[H] \vdash \texttt{let}\ x = M\ \texttt{in}\ N\ :\ T}\ let.$$

The crucial observation is that $[H_e, x \mapsto \Lambda\overline{v_m}.(\sigma_1, t_1)] \downarrow = (\delta_1,\ [\theta[H], x : \Pi\overline{w_l}.\delta_1(t_1)])$. But $S[H] = \rho_1 \circ \theta \circ \theta[H] = S[\theta[H]]$. We can therefore eliminate θ by absorbing it into S: $S[H, x : \Pi\overline{w_l}.\delta_1(t_1)] = S[\theta[H], x : \Pi\overline{w_l}.\delta_1(t_1)]$. Thus by inductive hypothesis on the second premise we have

$$W_\Pi(H_e, x \mapsto \Lambda\overline{v_m}.(\sigma_1, t_1);\ N) = \Lambda\overline{u_n}.(\sigma_2, t_2).$$

Let $[H_e, x \mapsto \Lambda\overline{v_m}.(\sigma_1, t_1), y \mapsto \Lambda\overline{u_n}.(\sigma_2, t_2)] \downarrow = (\delta_2, H_2)$, $\theta_2 \circ \delta_1 = \delta_2$, and $H_2(y) = \Pi\overline{z_k}.t_2$. The inductive hypothesis also gives a ρ_2 such that $\rho_2 \circ \theta_2 = S$ and $\rho_2[\Pi\overline{z_k}.t_2] = T$.

Now, $join(\sigma_2, \sigma_1) = \sigma$ succeeds since δ_2 exists (δ_2 is an instance of σ_2 and σ_1), and so

$$W_\Pi(H_e;\ let\ x = M\ in\ N) = \Lambda\overline{u_n}\Lambda\overline{v_m}.(\sigma, \sigma(t_2))$$

succeeds. We also have $[H_e, y \mapsto \Lambda\overline{u_n}\Lambda\overline{v_m}.(\sigma, \sigma(t_2))] \downarrow = (\delta_2, H_3)$, and we know that $H_3(y) = \Pi\overline{z_k}.\delta_2(t_2)$. Now $\theta_2 \circ \theta \circ \delta = \delta_2$ and $\rho_2 \circ (\theta_2 \circ \theta) = S \circ \theta = S$. Finally, $\delta_2(t_2) = t_2$ by definition of δ_2, and so

$$\rho_2[\Pi\overline{z_k}.\delta_2(t_2)] = \rho_2[\Pi\overline{z_k}.t_2] = T.$$

□

[4] Leivant defined inference rules for a more general set of constructors than `abs` and `app` (but without `let`). He also does not use generic applications.

Corollary 2.1 *(Completeness of W_Π)*
If for a closed type environment H such that $H \vdash M : T$, then $W_\Pi(H\uparrow; M) = \Lambda\overline{v_m}.(\sigma, t)$ such that T is an instance of $\Pi\overline{v_m}.t$.

Proof: We may assume without loss of generality that $\overline{v_m}$ are distinct from all variables in H. Set $S = \sigma$. It follows easily from the definition of the algorithm that σ does not contain variables other than $\overline{v_m}$ in its support. Thus $S[H] = H$. Similarly from the definition of the algorithm, $\sigma(t) = t$. In terms of the above theorem, here $\delta = \emptyset$ and $\delta' = \sigma$, so we set $\rho = \emptyset$ and the corollary follows. □

The $\downarrow$ operation is not needed in the algorithm for closed type environments since in the returned substitution all fugitives are captured. If the environment can be initially open, then we must free the innocent variables from bondage. The *generalized* W_Π algorithm merely requires a simple extra step: Let $W_\Pi(H\uparrow;\ M) = \Lambda\overline{v_m}.(\sigma, t)$. Then $[H\uparrow, y \mapsto \Lambda\overline{v_m}.(\sigma, t)]\downarrow = (\sigma, H')$. Return $H'(y)$. It will follow that $H' \vdash M : H'(y)$.

4 Related Work

The technique presented here is related to the work of Leivant [9] and of Appel and Shao [1] in type inferencing with multi-environments (environments where variables map to sets of types). Leivant's algorithm "V" returns a multi-environment (or multi-base) and a type given a program expression. Type inference in algorithm V does not take place under a given type environment. As a consequence, there is nothing to constrain the generalization of free type variables. Variables can be given multiple instantiations which are then resolved at the end. But algorithm V does not include a case for `let`. Leivant chose to address `let` polymorphism in the context of a *rank 2 conjunctive type discipline.* Wand [13] gave a similar algorithm, which likewise bypassed `let`. Appel and Shao's algorithm W^* [1] can be seen as essentially an extension of algorithm V to include `let`. They use a procedure called *Monounify* which serves basically the same purpose as *join*. W^* is similar to the approach here in that it too does not use *gen* (*gen* would be meaningless since there is no environment in the input to W^*). Instead, for the `let` case W^* uses a complicated operation called *Polyunify,* which generates a new *set* of copies of multi-environments (or "assumption environments") for every occurrence of the `let`-bound variable. The *Polyunify* technique is a "brute force" method akin to replacing *let* $x = M$ *in* N with $N[M/x]$. The multi-environment returned by W^* can be enormous, and will have to be further resolved with a given type environment (using their *Match* procedure) to derive the final type. Because of this complexity, Appel and Shao themselves favored a customization of Kaes' algorithm "D" [8] for their purpose of *smartest recompilation.* Furthermore, the correctness of W^* was proved by a complicated reduction to the correctness of algorithm W, and not to the Damas-Milner typing discipline itself. In fairness, the algorithm W^* allows concurrent type inference in the `let` case, while in algorithm W_Π only the case for $(M\ N)$ allows for parallel computation.

The motivation for W^* was to support separate compilation, where the types of program variables are not always available. Each program variable is always eagerly given the most general type (a free type variable), and the various possible instantiations are resolved when the type is finally known. The algorithm W_Π as given already contains the essential components necessary for this purpose. We can assign to each program variable that is not contained in the known type environment the most general type scheme $\Pi v.v$. Then W_Π will return a substitution containing the different possible instantiations of v. For example, assume that the type of f is unknown. Consider the expression $let\ x = (f\ 2)\ in\ (f\ 2.5)$. If f is mapped to $\Lambda v.(\emptyset, v)$, then W_Π will return the structure

$$\Lambda b \Lambda c \Lambda v_1 \Lambda v_2.([real \rightarrow c/v_2, int \rightarrow b/v_1],\ c).$$

If we knew that the variables v_1 and v_2 are in fact copies of the type scheme $\Pi v.v$, then we can infer the correct type for the expression once the type of f is available. Assume we now know that the type of f is actually $\Pi v.v \rightarrow v$. We can apply Appel and Shao's *Match* technique to the two instantiations $real \rightarrow c$ and $int \rightarrow b$ with two separate instances of $\Pi v.v \rightarrow v$: $\Pi u.u \rightarrow u$ and $\Pi w.w \rightarrow w$. This will reveal that $c = real$ and $b = int$, and therefore *real* should be the type for $let\ x = (f\ 2)\ in\ (f\ 2.5)$. To implement this technique correctly, we must modify W_Π so that we can identify which variables are in fact copied from type schemes $\Pi v.v$ associated with undeclared program variables. One approach may be to label these special type variables with the program variable they are associated with. This approach would be similar to Appel and Shao's adaptation of Kaes' algorithm D for *constrained* types [8]. However, algorithm D again uses the gen operation in the `let` case. Appel and Shao's proof of the correctness of their version of D is again by reduction to algorithm W, and not to the proof-theoretic typing discipline.

The origin of the eager quantification technique came from trying to implement type inference in a higher-order logic programming language. Such a declarative treatment will aid the analysis of functional languages in the context of *logical frameworks,* such as the dependent-type calculus LF [6]. The desire here is for an executable proof-theoretic formulation of type inference. That is, type inference should be presentable as proof search. The original Damas-Milner calculus is too nondeterministic for this purpose. Previous attempts at its alteration either took short-cuts with the `let` case or were stopped by gen. In [4], Hannan gave proof-theoretic formulations of the natural semantics of ML. But his technique for `let` was basically to replace $let\ x = M\ in\ N$ with $N[M/x]$. To allow `let`-expressions to be typed naturally, Harper defined in [5] an "algorithmic" version of the Damas-Milner calculus for the express purpose of allowing the modified typing rules of the new calculus to become logic programs that yield principal type schemes. He defined a predicate called *witnessed* that captures the maximality condition implemented by gen. Application of the gen operation is replaced by proving that a type scheme is *witnessed.* Specifying the *witnessed* predicate directly as logic programming, however, requires a forward-chaining operation which is inconsistent with the goal-directed nature of logic-programming interpreters. Another problem with type inference was the need for an inexhaustible supply of new variables. In the context of "meta-programming in logic," one can either use the meta-logic's inherent "logic variables" or

define data structures such as strings to represent object-level variables. Using the meta-logic's own variables (called the "non-ground representation") is only adequate for a very small range of problems[5]. Strings and similar structures are too algorithmic and "low level." The author wished to use λ-bound variables in the simply typed λ-calculus (i.e, higher-order abstract syntax) to represent the new type variables that are unavoidable for type inference. But λ-abstraction is also the most natural representation for type quantification. Thus both free *and* bound type variables are represented in the meta-logic as λ-bound variables. This *uniform treatment* of type variables led to the eager type scheme method: even variables that are "supposed to be free" are bound. An implementation of a slight variation of the W_Π algorithm has been given in the logic programming language L_λ [11] without using any extra-logical extensions. This implementation is described in the author's Ph.D. thesis [10].

Conclusion

A common problem of all the related work we've examined is their difficulty with the `let` case. The traditional *gen* operation created an undesirable gulf between the declarative typing discipline on the one side and the algorithmic type inferencing process on the other. With algorithm W_Π we have significantly narrowed this gulf. With the basic technique of eager polymorphism we also hope to provide a new, computationally sound starting point from which various issues of type inference can be studied. It of course remains to extend W_Π to the full ML language. Another area we are investigating concerns the use of eager polymorphism with respect to polymorphic references. It is hoped that we will be able to accept more type-safe programs than current methods. The W_Π algorithm can also lead to the early reportage of typing errors. Because substitutions are composed instead of joined in algorithm W, by the time we discover a type error the substitutions may have obscured its origin. We would like to be able to inform the programmer which section of the code originated the type error. Combined with a constrained typing discipline, the W_Π technique can potentially offer a new solution to this problem. We also hope to study the eager type scheme technique in the context of typing disciplines other than ML polymorphism.

Acknowledgments

This research is supported in part by the following grants: ONR N00014-93-1-1324, NSF CCR-91-02753, NSF CCR-92-09224, and DARPA N00014-85-K-0018. The author wishes to thank Dale Miller, Philip Wickline, and especially Sandip Biswas for support and invaluable discussions.

[5] See [7, 10] for further discussion of issues in meta-programming in logic.

References

[1] Andrew Appel and Zhong Shao. Smartest Recompilation. In *Tenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1993. Longer version as Princeton University Technical Report CS-TR-395-92.

[2] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Ninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[3] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press, 1992.

[4] John Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152, April 1993.

[5] Robert Harper. Systems of polymorphic type assignment in LF. Technical Report CMU-CS-90-144, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1990.

[6] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[7] P. M. Hill and J. G. Gallagher. Meta-programming in logic programming. Technical Report Report 94.22, University of Leeds, hill@scs.leeds.ac.uk, August 1994. To appear in Vol. 5 of the *Handbook of Logic in Artificial Intelligence and Logic Programming,* Oxford University Press.

[8] Stefan Kaes. Type Inference in the presence of Overloading, Subtyping, and Recursive types. In *1992 ACM conference on LISP and Functional Programming, San Francisco, CA*, pages 193–204. ACM Press, 1992.

[9] Daniel Leivant. Polymorphic type inference. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 88–98, 1983.

[10] Chuck Liang. *Substitution, Unification and Generalization in Meta-Logic.* PhD thesis, University of Pennsylvania, September 1995.

[11] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[12] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[13] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Infomaticae*, 10:115–122, 1987.