Quantifier elimination and parametric polymorphism in programming languages

*Harry G. Mairson**
Department of Computer Science
Brandeis University
Waltham, Massachusetts 02254

February 5, 1996

Abstract

We present a simple and easy to understand explanation of ML type inference and parametric polymorphism within the framework of type monomorphism, as in the first order typed lambda calculus. We prove the equivalence of this system with the standard interpretation using type polymorphism, and extend the equivalence to include polymorphic fixpoints. The monomorphic interpretation gives a purely combinatorial understanding of the type inference problem, and is a classic instance of quantifier elimination, as well as an example of Gentzen-style cut elimination in the framework of the Curry-Howard propositions-as-types analogy.

*Supported by NSF Grant CCR-9017125, and grants from Texas Instruments and from the Tyson Foundation.

1 Introduction

In his influential paper, "A theory of type polymorphism in programming," Robin Milner proposed an extension to the first order typed λ-calculus which has become known as the core of the ML programming language [Mil78, HMT90]. The extension augmented the monomorphic type language of the first order typed λ-calculus with *polytypes* (also known as *type schemes*) allowing a limited form of quantification over type variables. The expression language was similarly expanded by introducing the construct $\texttt{let}\ x = E\ \texttt{in}\ B$, where by typing E with a polytype, the free occurances of x in B could be typed *differently* (i.e., *polymorphically*) by varied instantiations of the quantified variables in the polytype. The added expressiveness of the type language then allowed $\texttt{let}\ x = E\ \texttt{in}\ B$ to be typable where the λ-calculus "equivalent" $(\lambda x.B)E$ might not be; the classic example of this facility is that $\texttt{let}\ I = \lambda z.z\ \texttt{in}\ II$ is typable in ML, while $(\lambda I.II)(\lambda z.z)$ is not first order typable.

Type polymorphism has since been incorporated into a variety of functional programming languages [HW88, Tur85]. Among its virtues are *static typing*, so that all typing is done at compile time, with the guarantee that typechecked programs will not "go wrong" at run time; *parametric polymorphism*, so that polymorphically-typed code can be reused (via `let`) on abstract data types; and *decidable type inference*, where the compiler can automatically infer the most general type information (the so-called *principal type*) for an expression, so that any typing for the expression is a *substitution instance* of the principal type.

To what extent is Milner's proposal of type polymorphism necessary to achieve this degree of parametric polymorphism? Surprisingly, the type language of the first order typed λ-calculus is sufficient to support ML-style parametric polymorphism, as long as we use the following inference rule for typing `let`-expressions:

$$(\texttt{let}) \qquad \frac{\Gamma \rhd E{:}\,\tau_0 \qquad \Gamma \rhd [E/x]B{:}\,\tau_1}{\Gamma \rhd \texttt{let}\ x = E\ \texttt{in}\ B{:}\,\tau_1}$$

Any ML program without free variables that is typable in the standard Milner-Damas inference system [DM82] is also typable using the classical Curry inference system [CF58] augmented with the above rule. Hence parametric polymorphism as realized in ML may be achieved within the framework of type monomorphism.

Observe that the above inference rule realizes parametric polymorphism ("code reuse") explicitly through the expression $[E/x]B$: namely, each free occurance of x gets replaced with a separate *copy* of the program E. The example of typing $\texttt{let}\ I = \lambda z.z\ \texttt{in}\ II$, for instance, is reduced to typing $(\lambda z.z)(\lambda z.z)$, so each $\lambda z.z$ may be typed *differently*. The effect is the same as considering the expression to be a *marked* redex $\underline{(\lambda I.II)(\lambda z.z)}^1$ in the theory of labelled reductions [Bar84].

The monomorphic realization of ML's parametric polymorphism is not new. A recent survey of type systems in programming [Mit90] attributes the observation to Albert Meyer. An earlier appearance of the idea is found in the dissertation of Luis Damas [Dam85], and in fact a question about it is found in the 1985 postgraduate examination in computing at Edinburgh University [Edi88].

In this paper, we present a simple and easy to understand explanation of ML type inference in the framework of type monomorphism, where we prove its equivalence to the standard interpretation using type polymorphism. In addition, we analyze an extension of the ML inference system proposed by Alan Mycroft [Myc84], allowing fixpoints where the variable appearing in a recursion equation may have a polymorphic type. While type inference for this system is not computable [KTU90], we show that the inference system has nonetheless a purely monomorphic interpretation.

We believe that the monomorphic interpretation is important because it gives a *purely combinatorial* understanding of a significant fragment of the Girard/Reynolds second-order polymorphic typed λ-calculus [Gir72, Rey74]. It provides as well a classic example of *quantifier elimination*, which in the context of the Curry-Howard propositions-as-types analogy serves as a sort of Gentzen-style cut elimination. The simple combinatorics of the monomorphic interpretation, which reduces the problem of type inference to first-order unification [Rob65], has played a central role in a complete analysis of the computational complexity of ML type inference [KM89, Mai90, KMM91], as well as providing insight in the first significant lower bounds on type inference for higher-order typed λ-calculi [HM91].

2 Preliminaries

2.1 Expressions

We consider ML expressions defined by the grammar:

$$\mathcal{E} \ ::= \ x \mid \mathcal{E}\mathcal{E} \mid \lambda x.\mathcal{E} \mid \texttt{let}\ x = \mathcal{E}\ \texttt{in}\ \mathcal{E} \mid \mathbf{fix}\, x.\mathcal{E}$$

where x ranges over a set $\mathcal{V}$ of *expression variables*. Excluding expressions of the form $\mathbf{fix}\, x.E$ where $E \in \mathcal{E}$, the language considered is known as *Core ML* (see, for example, [MH88]). The syntax of Core ML is just that of the λ-calculus augmented with the polymorphic `let` construct. We write $FV(E)$ to denote the *free variables* of E. We allow α-renaming and β-reduction as in the λ-calculus, as well as reduction of `let`-expressions following the rule:

$$\texttt{let}\ x = E\ \texttt{in}\ B \ \longrightarrow_{\texttt{let}}\ [E/x]B$$

For more details concerning reductions in the λ-calculus and ML, see [Bar84, HS87, HMT90].

2.2 Types

The syntax of *types* is given by the grammar:

$$\begin{aligned} \mathcal{T}_0 \ &::= \ t \mid \mathcal{T}_0 \to \mathcal{T}_0 \\ \mathcal{T} \ &::= \ \mathcal{T}_0 \mid \forall t.\mathcal{T} \end{aligned}$$

where t ranges over a set $\mathcal{TV}$ of *type variables*. We refer to $\tau \in \mathcal{T}_0$ as *monotypes*, and $\sigma \in \mathcal{T}$ as *polytypes* (sometimes also called *type schemes*).

We define a partial order $\preceq$ on $\mathcal{T}_0$ as $\tau_1 \preceq \tau_2$ iff there exists a substitution $\Sigma: \mathcal{TV} \longrightarrow \mathcal{T}_0$ such that $\Sigma\tau_1 \equiv \tau_2$, where $\equiv$ denotes syntactic equivalence (overloaded for use on expressions as well). We interpret polytypes as sets of monotypes, using the following interpretation:

$$\begin{array}{rcll} \langle\alpha\rangle & = & \{\alpha\} & \text{where } \alpha \text{ is a monotype} \\ \langle\forall t.\alpha\rangle & = & \bigcup_{\tau\in\mathcal{T}_0} \langle[\tau/t]\alpha\rangle, & \end{array}$$

where $[\tau/t]\alpha$ denotes the substitution of τ for free occurances of t in α.

The interpretation of polytypes as sets of monotypes allows the definition of a partial order $\sqsubseteq$ on $\mathcal{T}$: we write $\sigma_1 \sqsubseteq \sigma_2$ iff $\langle\sigma_2\rangle \subseteq \langle\sigma_1\rangle$. It is easy to see, for instance, that $\forall t.\alpha \sqsubseteq [\tau/t]\alpha$ for any polytype α and monotype τ. Note the minimal element of $\mathcal{T}$ is $\forall t.t$, since $\langle\forall t.t\rangle = \mathcal{T}_0$. We further define an equivalence relation on $\mathcal{T}$ as $\sigma_1 \cong \sigma_2$ iff $\langle\sigma_2\rangle = \langle\sigma_1\rangle$, and write $[\sigma]$ to denote the equivalence class $\{\alpha\colon\ \alpha \cong \sigma\}$. (This equivalence class definition will be used when we wish to argue that the names and order of bound variables in a polytype are not significant.) When σ is a polytype, we write $\overline{\sigma}$ to denote the monotype derived by removing all quantifiers from σ; when τ is a monotype, we write $\vec{\forall}.\tau$ to denote the polytype derived by quantifying over some subset of type variables in τ.

2.3 Inference rules

Expressions are associated with types using a fixed set of *inference rules.* We describe two such systems of rules: the first being the standard one given by Damas and Milner [DM82] which we call the *polytype system*, and the second one a variant called the *monotype system.* As its name suggests, the monotype system associates expressions with monotypes only. The major point of this paper is to show simply why this limitation is not truly a restriction.

The inference rules manipulate an expression called a *type judgement,* written $\Gamma \ \triangleright\ E\colon \sigma$, where $E \in \mathcal{E}$, $\sigma \in \mathcal{T}$, and $\Gamma\colon\ FV(E) \longrightarrow \mathcal{T}$. The type judgement is read as "with environment (context) Γ, expression E has type σ." In the λ-calculus, environments associate values to free variables in an expression, while in this case the environment is used to associate *types* with the free variables.

We give below the inference rules for the polytype and monotype systems. The polytype system is due to [DM82], and the monotype system is essentially due to [CF58] augmented with the rule for `let`. Observe the use in rule ($\texttt{let}_P$) of types with quantifiers (namely, the binding for x), requiring the rules (gen_P) and ($inst_P$) for quantifier introduction and elimination. For more details on type inference rules, we recommend [Car84, Han87, Mil78, Wan87].

2.3.1 Core ML inference rules for the polytype system

(var_P) $$\frac{}{\Gamma \cup \{x\colon \sigma\} \ \rhd \ x\colon \sigma}$$

(gen_P) $$\frac{\Gamma \ \rhd \ E\colon \sigma(t) \qquad [t \notin FV(\Gamma)]}{\Gamma \ \rhd \ E\colon \forall t.\sigma(t)}$$

$(inst_P)$ $$\frac{\Gamma \ \rhd \ E\colon \forall t.\sigma(t)}{\Gamma \ \rhd \ E\colon [\tau/t]\sigma}$$

(abs_P) $$\frac{\Gamma \cup \{x\colon \tau_0\} \ \rhd \ E\colon \tau_1}{\Gamma \ \rhd \ \lambda x.E\colon \tau_0 \to \tau_1}$$

(app_P) $$\frac{\Gamma \ \rhd \ M\colon \tau_0 \to \tau_1 \qquad \Gamma \ \rhd \ N\colon \tau_0}{\Gamma \ \rhd \ MN\colon \tau_1}$$

$(\mathtt{let}_P)$ $$\frac{\Gamma \ \rhd \ E\colon \sigma \qquad \Gamma \cup \{x\colon \sigma\} \ \rhd \ B\colon \tau}{\Gamma \rhd \mathtt{let}\ x = E\ \mathtt{in}\ B\colon \tau}$$

2.3.2 Core ML inference rules for the monotype system

(var_M) $$\frac{}{\Gamma \cup \{x\colon \tau\} \ \rhd \ x\colon \tau}$$

(app_M) $$\frac{\Gamma \ \rhd \ M\colon \tau_0 \to \tau_1 \qquad \Gamma \ \rhd \ N\colon \tau_0}{\Gamma \ \rhd \ MN\colon \tau_1}$$

(abs_M) $$\frac{\Gamma \cup \{x\colon \tau_0\} \ \rhd \ E\colon \tau_1}{\Gamma \ \rhd \ \lambda x.E\colon \tau_0 \to \tau_1}$$

$(\mathtt{let}_M)$ $$\frac{\Gamma \ \rhd \ E\colon \tau_0 \qquad \Gamma \ \rhd \ [E/x]B\colon \tau_1}{\Gamma \rhd \mathtt{let}\ x = E\ \mathtt{in}\ B\colon \tau_1}$$

3 Equivalence of the polytype and monotype systems

What should it mean when we say that the monotype system is "equivalent" to the polytype system? A first guess might be that for any expression E, monotype τ, and context Γ, $\vdash_P \Gamma \rhd E\colon \tau$ iff $\vdash_M \Gamma \rhd E\colon \tau$. However, if Γ contains polytypes, it is not a valid context for a monotype judgement. Unfortunately, if we try $\vdash_P \Gamma \rhd E\colon \tau$ iff $\vdash_M \Gamma_0 \rhd E\colon \tau$ where Γ_0 are the monotype bindings of Γ, the statement is not true: consider $\vdash_P \{I\colon \forall t.t \to t\} \rhd II\colon t \to t$ iff $\vdash_M \emptyset \rhd II\colon t \to t$ — the monotype judgement is clearly false.

A second guess might be to insist that E be a closed term. A proof along these lines can indeed be given, where we proceed by a double induction on the structure of E and the maximum number of `let`-reductions needed to reduce E to `let`-normal form; see the Appendix of [KMM91]. However, the proof is overly tedious and technical, and requires an understanding of minimal complete developments in the λ-calculus [HS87, Bar84]. But most of all, it contradicts an overwhelming sentiment that the equivalence we want is something very *simple* which should be *easy* to prove. What, then, should the equivalent of $\vdash_P \{I{:}\,\forall t.t \to t\} \rhd II{:}\,t \to t$ be in the monotype system? (Note II is not closed.) We propose the following: the monotype equivalent should be $\vdash_M \emptyset \rhd [E/I]II{:}\,t \to t$, where $[E/I]II$ is a closed term, and $\vdash_P \emptyset \rhd E{:}\,\forall t.t \to t$. Of course, we are thinking in this case of $E \equiv \lambda z.z$.

What justifies this specification of equivalence? Polytypes are a kind of *shorthand* in the spirit of "code reuse" and the Gentzen "cut." In identifying an expression variable with a polytype, there is an implicit assumption that a piece of code exists with that type; what we have done in this example is simply to *insert* the code in place of its variable representative.

Generalizing from the example, we propose the following as a reasonable definition of "equivalence:"

Definition 3.1 *Let* $\Gamma = \{w_1{:}\,\alpha_1, w_2{:}\,\alpha_2, \ldots, w_m{:}\,\alpha_m\}$ *be any context of monotype bindings, and* $\Gamma' = \{y_1{:}\,\beta_1, y_2{:}\,\beta_2, \ldots, y_n{:}\,\beta_n\}$ *be any context of polytype bindings. Let* $\mathcal{F} = \{F_1, F_2 \ldots, F_n\}$ *be a set of terms where* β_j *is the principal type of* F_j*; specifically, we insist that for* $1 \le j \le n$*,*

$$\begin{array}{ll} \vdash_P & \Gamma \cup \{y_1{:}\,\beta_1, y_2{:}\,\beta_2, \ldots, y_{j-1}{:}\,\beta_{j-1}\} \rhd F_j{:}\,\beta_j \\ \vdash_M & \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_{j-1}/y_{j-1}]F_j{:}\,\overline{\beta_j} \end{array}$$

where $\overline{\beta_j}$ *is* β_j *with all quantifiers removed.*

Given this framework, we can precise what is meant by equivalence. Let E *be any term and* τ *be any monotype; then*

$$\vdash_P \Gamma \cup \Gamma' \rhd E{:}\,\tau \qquad \textit{if and only if} \qquad \vdash_M \quad \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]E{:}\,\tau \tag{1}$$

It is clear, and indeed natural, that the equivalent monotype judgement should be contingent on the explicit substitution of code represented at the type level by polytypes. In the case of a closed term E with empty contexts, we have $\vdash_P \emptyset \rhd E{:}\,\tau$ iff $\vdash_M \emptyset \rhd E{:}\,\tau$, as in [KMM91]. However, inspired by the example of Tait's strong normalization theorem for the first order typed λ-calculus [Tai67], we have facilitated the proof by strengthening the induction hypothesis of what is to be a syntax directed induction on E.

Before continuing with the proof, we introduce a standard structural lemma allowing us to "normalize" derivations in the polytype system for use in a syntax-directed proof.

Lemma 3.2 *Let* $\Pi \vdash_P \Gamma \rhd E{:}\,\vec{\forall}.\tau$ *where* Γ *is a context,* τ *is a monotype, and* $\vec{\forall}$ *denotes a (possibly empty) list of quantified variables. Then if* E *is not a variable, there exists a proof* $\Pi' \vdash_P \Gamma \rhd E{:}\,\tau$ *where the last rule used in* Π' *is either* (var_P)*,* (abs_P)*,* (app_P)*, or* $(\mathtt{let}_P)$*.*

Proof. Observe that $\Pi \vdash_P \Gamma \rhd E{:}\,\vec{\forall}.\tau$ is a syntax directed proof except for the use of (gen_P) and $(inst_P)$. The lemma states that the final uses of (gen_P) and $(inst_P)$ can be removed. The proof proceeds by induction on the number of such uses; in the basis case, clearly $\vec{\forall}.\tau \equiv \tau$.

For the inductive step, we must consider only two cases:

Case 1: The proof Π ends using the rule (gen_P):

$$(gen_P) \qquad \frac{\Gamma \ \triangleright \ E{:}\vec{\forall}.\tau(t)}{\Gamma \ \triangleright \ E{:}\forall t.\vec{\forall}.\tau(t)}$$

Simply remove the last step of the proof to remove one quantifier from the type, and apply the inductive hypothesis.

Case 2: The proof Π ends using the rule $(inst_P)$:

$$(inst_P) \qquad \frac{\Gamma \ \triangleright \ E{:}\forall t.\vec{\forall}.\tau(t)}{\Gamma \ \triangleright \ E{:}\vec{\forall}.\tau(\alpha)}$$

where α is a monotype. Observe that the last rules appearing in the proof are a series of uses of (gen_P) and $(inst_P)$, where the former adds a quantifier, and the latter removes a quantifier. As such, they act like a stack. Identify the point (I) in the proof where t is universally quantified:

$$(gen_P) \qquad \frac{\Gamma \ \triangleright \ E{:}\vec{\forall}.\tau'(t)}{\Gamma \ \triangleright \ E{:}\forall t.\vec{\forall}.\tau'(t)}$$

We now proceed as follows:

1. In the subproof rooted at (I), replace all free occurences of t by α;
2. In the deductions from (I) until the end of the proof, remove the binding $\forall t$, replacing newly free occurences of t by α;
3. Remove the conclusions of (I) and the final inference. The proof now has two fewer uses of (gen_P) and $(inst_P)$, so we can apply the inductive hypothesis.

■

Given the lemma and the stated assumptions on Γ, Γ', and $\mathcal{F}$, we prove the above statement (1) in Definition 3.1 via structural induction on E, proceeding by case analysis. We assume by renaming that no variable is bound or quantified more than once.

Case $E \equiv w_i$

Then necessarily $\tau \equiv \alpha_i$ and $\vdash_P \Gamma \cup \Gamma' \triangleright w_i{:}\,\alpha_i$; since $[F_1/y_1][F_2/y_2]\cdots[F_n/y_n]w_i \equiv w_i$, the result is immediate.

Case $E \equiv y_j$

In the forward direction, assume $\vdash_P \Gamma \cup \Gamma' \rhd y_j{:}\,\tau$. Since $\vdash_P \Gamma \cup \Gamma' \rhd y_j{:}\,\beta_j$ is a principal typing, we know $\beta_j \sqsubseteq \tau$; since $\tau \in \mathcal{T}_0$, we know there exists a substitution Σ such that $\Sigma\overline{\beta_j} = \tau$. But since $[F_1/y_1][F_2/y_2]\cdots[F_n/y_n]y_j \equiv [F_1/y_1][F_2/y_2]\cdots[F_{j-1}/y_{j-1}]F_j$, and

$$\Pi \vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_{j-1}/y_{j-1}]F_j{:}\,\overline{\beta_j}$$

we know

$$\vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_{j-1}/y_{j-1}]F_j{:}\,\tau$$

by applying Σ to all types appearing in the proof Π.

In the reverse direction, suppose $\vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_{j-1}/y_{j-1}]F_j{:}\,\tau$; then by principality we know $\Sigma\overline{\beta_j} = \tau$. Given $\vdash_P \Gamma \cup \Gamma' \rhd y_j{:}\,\beta_j$, we derive $\vdash_P \Gamma \cup \Gamma' \rhd y_j{:}\,\tau$ by instantiating the $\forall$-bound variables of β_j according to Σ.

Case $E \equiv GH$

To prove "only if," if $\vdash_P \Gamma \cup \Gamma' \rhd GH{:}\,\tau$, then $\vdash_P \Gamma \cup \Gamma' \rhd G{:}\,\tau' \to \tau$ and $\vdash_P \Gamma \cup \Gamma' \rhd H{:}\,\tau'$ for some monotype τ', by Lemma 3.2. From induction on G and H, we know $\vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]G{:}\,\tau' \to \tau$ and $\vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]H{:}\,\tau'$, so the result follows by use of (app_M). Note the implications are all reversible, except that Lemma 3.2 is not needed.

Case $E \equiv \lambda x.G$

To prove "only if," if $\vdash_P \Gamma \cup \Gamma' \rhd \lambda x.G{:}\,\tau' \to \tau''$ where $\tau \equiv \tau' \to \tau''$, then by Lemma 3.2 we know $\vdash_P \Gamma \cup \{x{:}\,\tau'\} \cup \Gamma' \rhd G{:}\,\tau''$. By induction on G (with a larger monotype context, since the binding for x is added), we have $\vdash_M \Gamma \cup \{x{:}\,\tau'\} \rhd [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]G{:}\,\tau''$, and by (abs_M), $\vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]\lambda x.G{:}\,\tau' \to \tau''$. Again, all implications are reversible.

Case $E \equiv \texttt{let}\ y = G\ \texttt{in}\ H$

If $\vdash_P \Gamma \cup \Gamma' \rhd \texttt{let}\ y = G\ \texttt{in}\ H{:}\,\tau$, then by Lemma 3.2 and $(\texttt{let}_P)$, $\vdash_P \Gamma \cup \Gamma' \rhd G{:}\,\sigma$ for (principal) polytype σ, and $\vdash_P \Gamma \cup \Gamma' \cup \{y{:}\,\sigma\} \rhd H{:}\,\tau$. By $(inst_P)$, $\vdash_P \Gamma \cup \Gamma' \rhd G{:}\,\overline{\sigma}$, so by induction on G we have a proof

$$\Pi \vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]G{:}\,\overline{\sigma}.$$

Hence by induction on H with polytype context $\{y_1{:}\,\beta_1, y_2{:}\,\beta_2, \ldots, y_n{:}\,\beta_n, y{:}\,\sigma\}$ and associated code $\{F_1, F_2, \ldots, F_n, G\}$, we have $\vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_n/y_n][G/y]H{:}\,\tau$, which we rewrite as $\vdash_M \Gamma \rhd [[F_1/y_1][F_2/y_2]\cdots[F_n/y_n]G/y][F_1/y_1][F_2/y_2]\cdots[F_n/y_n]H{:}\,\tau$. Using proof Π above and $(\texttt{let}_M)$, we then have a proof of

$$\vdash_M \Gamma \rhd \texttt{let}\ y = [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]G\ \texttt{in}\ [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]H{:}\,\tau,$$

which is syntactically identical to

$$\vdash_M \Gamma \rhd [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]\texttt{let}\ y = G\ \texttt{in}\ H{:}\,\tau.$$

Once again, the argument is reversible.

Given our above definition of equivalence, we then have:

Theorem 3.3 *The polytype and monotype inference systems for Core ML are equivalent.*

Corollary 3.4 *Let E be a closed term and τ be a monotype. Then $\vdash_P \emptyset \rhd E\colon \tau$ if and only if $\vdash_M \emptyset \rhd E\colon \tau$.*

3.1 Parametric polymorphism, cut elimination, and proof theory

In the well known Curry-Howard propositions-as-types analogy, we read $E\colon \sigma$ not as "expression E has type σ," but "expression E is a proof of proposition σ." In this case, the function $\rightarrow$ in types is read as logical implication. An environment Γ then serves as a set of labelled assumptions, so that a type judgement $\Gamma \rhd E\colon \sigma$ elaborates a classical *sequent* $\tilde{\Gamma} \vdash \sigma$.

Proofs in sequent calculus, like type derivations, can be written in the form of trees, where the leaves form propositional hypotheses. The logical formalism of *cancelling hypotheses* via $\rightarrow$-introduction is reflected in removing type assumptions and introducing λ-abstraction. (For a further detailed but elementary discussion, see [vanD79].)

The process of β-reduction in the simply-typed λ-calculus can be interpreted as a transformation on proof trees: if $\Pi_M \vdash \Gamma \rhd \lambda x.M\colon \tau_1 \rightarrow \tau_2$ and $\Pi_N \vdash \Gamma \rhd N\colon \tau_1$, by "modus ponens" we get $\Pi_{MN} \vdash \Gamma \rhd (\lambda x.M)N\colon \tau_2$. In this case we also know $\vdash \Gamma \rhd [N/x]M\colon \tau_2$, since $\Pi'_M \vdash \Gamma \cup \{x\colon \tau_1\} \rhd M\colon \tau_2$; we replace the *assumption* $x\colon \tau_1$ (appearing as a *leaf* in the proof tree Π'_M) by the *subtree (proof)* Π_N. Interpreted at the proof theory level, this transformation is an example of what is called, after Gentzen [Gen69], *cut elimination*, since $(\lambda x.M)N$ represents a proof where τ_1 is proved *once*, a "shortcut" over $[N/x]M$, which may require many proofs of τ_1.

The parametric polymorphism in Core ML introduced by the `let` construct can be viewed as a more powerful form of cut-elimination. The cut-elimination via β-reduction allows one proof of a $\forall$-free proposition to be used several times, while cut-elimation via `let`-reduction allows one proof of a proposition to be used several times, provided that the "use" is always monomorphic ($\forall$-free). Rather than prove $P = \{\alpha \rightarrow \alpha, \beta \rightarrow \beta, (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)\}$, for example, we construct one proof of $\forall t.t \rightarrow t$, and instantiate t appropriately. The propositions in P have a *most general unifier*, namely a proposition π such that $\pi \preceq p$ for each $p \in P$. We make the related observation that the monomorphic inference rules for Core ML show that the *principal type property* proved in [Mil78] is a straightforward consequence of the existence of most general unifiers in the first-order domain.

Similar to β-reduction, `let`-reduction can be viewed as a proof transformation. Since the expression `let` $x = E$ `in` B may have a polytype assigned to x, each use of E in $[N/x]B$ can instantiate the quantifiers differently. The "same" proof is recycled to generate structurally similar propositions.

4 A characterization of polymorphic recursion by monotypes

The inference rules we have described thus far for typing ML programs do not include a rule for typing fixpoints, and hence do not allow recursion. In ML, fixpoints are constrained to be monomorphic; as such, the polytype system is usually extended with

$$(\mathit{fix}_{ML}) \qquad \frac{\Gamma \cup \{x\colon\tau\} \;\triangleright\; E\colon\tau}{\Gamma \;\triangleright\; \mathbf{fix}\, x.E\colon\tau}$$

It is not difficult to show that when this rule is added to the polytype system, $\vdash_P \Gamma \;\triangleright\; \mathbf{fix}\, x.E\colon\tau$ iff $\vdash_P \Gamma \;\triangleright\; \lambda x.Eq\ E\, x\colon\tau$, where $Eq \equiv \lambda p.\lambda q.K\ p\,(\lambda r.K\ (r\,p)\,(r\,q))$, given the usual definition $K \equiv \lambda x.\lambda y.x$, since Eq has principal type $\forall t.t \to t \to t$. As a consequence, adding monomorphic fixpoint does not make type inference particularly more complex.

Alan Mycroft [Myc84] proposed a more powerful variant to the above rule, whereby $\mathbf{fix}$-bound variables could occur *polymorphically*:

$$(\mathit{fix}_{P}) \qquad \frac{\Gamma \cup \{x\colon\sigma\} \;\triangleright\; E\colon\sigma}{\Gamma \;\triangleright\; \mathbf{fix}\, x.E\colon\sigma}$$

In this rule, σ is a *polytype*. It has recently been shown by Kfoury, Tiuryn, and Urzyczyn that type inference in the presence of such a polymorphic fixpoint is undecidable [KTU90].

In this section, we show that polymorphic fixpoint can also be described using type monomorphism only. As the polymorphic inference system has been augmented with the rule (fix_P), we add the following rules to the monotype system:

$$(\bot_M) \qquad \frac{}{\Gamma \;\triangleright\; \mathbf{fix}\, x.x\colon\tau}$$

$$(\mathit{fix}_{M}) \qquad \frac{\Gamma \;\triangleright\; E_k\colon\tau \qquad \Gamma \;\triangleright\; E_{k+1}\colon\tau}{\Gamma \;\triangleright\; \mathbf{fix}\, x.E\colon\tau}$$

where for any term E with free variable x, we define

$$\begin{aligned} E_0 &\equiv \bot \equiv \mathbf{fix}\, x.x \\ E_1 &\equiv \mathtt{let}\ x_0 = E_0\ \mathtt{in}\ [x_0/x]E \\ &\cdots \\ E_{k+1} &\equiv \mathtt{let}\ x_k = E_k\ \mathtt{in}\ [x_k/x]E \end{aligned}$$

Henceforth, we refer to the polytype system as augmented with rule (fix_P), and the monotype inference systems as augmented with rules $(\bot_M)$and (fix_M). We observe that, properly speaking, (fix_M) is actually a rule *schema*, since its syntax varies with the integer k. However, it should be noted that *all* the inference rules are actually schemas! The monomorphic rules for typing polymorphic fixpoint have a simple explanation. An initial approximation $\bot\colon\forall t.t$ is made for the fixpoint, and the principal types of the E_k are repeatedly computed to better approximate the fixpoint until (possible) convergence.

To carry out this approach, we must show that for a given term E_k with principal type μ_k approximating the least fixpoint, and a known (type) fixpoint σ of $\mathbf{fix}\, x.E$ in the polytype system, it follows that the principal type μ_{k+1} of $E_{k+1} \equiv \mathtt{let}\ x_k = E_k\ \mathtt{in}\ [x_k/x]E$ always satisfies $\mu_k \sqsubseteq \mu_{k+1} \sqsubseteq \sigma$. Since there are (up to renaming of $\forall$-bound variables) only a finite number of types σ' satisfying $\sigma' \sqsubseteq \sigma$, we know by a pigeonhole argument that the sequence must converge. (We could instead show that the types form a complete partial order, from which convergence of the sequence is assured, but we prefer to proceed using a more combinatorial approach.)

We begin by indicating how polytype inferences can be derived from monotype inferences:

Proposition 4.1 *Let $\Gamma, \Gamma', \mathcal{F}$ be defined as in Definition 3.1. If*

$$\vdash_M \Gamma\ \triangleright\ [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]E_j\colon \overline{\mu}$$

is a principal *typing for $j \in \{k, k+1\}$, and E is* **fix***-free, then $\vdash_P \Gamma \cup \Gamma'\ \triangleright\ \mathbf{fix}\, x.E\colon \mu$.*

Proof. By Theorem 3.3, we know that

$$\vdash_P \Gamma \cup \Gamma'\ \triangleright\ \mathtt{let}\ x_k = E_k\ \mathtt{in}\ [x_k/x]E\colon \mu,$$

so that

$$\vdash_P \Gamma \cup \Gamma' \cup \{x\colon \mu\}\ \triangleright\ E\colon \mu,$$

and hence by rule (*fix*$_P$) ,

$$\vdash_P \Gamma \cup \Gamma'\ \triangleright\ \mathbf{fix}\, x.E\colon \mu.$$

∎

Observe that in the above Proposition, μ must be the *principal* type of E_k and E_{k+1}, as principality is required for the proof of Theorem 3.3. To prove the converse, namely that monotype inferences can be derived from polytype inferences, a bit more detail is required. We begin with the following simple observation:

Proposition 4.2 *If $\vdash_P \Gamma \cup \Gamma'\ \triangleright\ \mathbf{fix}\, x.E\colon \sigma$, then $\vdash_P \Gamma \cup \Gamma'\ \triangleright\ E_k\colon \sigma$ for all $k \geq 0$, and the* principal *type μ_k of E_k in environment $\Gamma \cup \Gamma'$ satisfies $\mu_k \sqsubseteq \sigma$.*

Proof. By induction on k. The case $k = 0$ is trivial. For $k \geq 0$, recall by (*fix*$_P$) that $\vdash_P\ \Gamma \cup \Gamma' \cup \{x\colon \sigma\}\ \triangleright\ E\colon \sigma$. To show the same judgement holds of E_{k+1}, observe that as $E_{k+1} \equiv \mathtt{let}\ x_k = E_k\ \mathtt{in}\ [x_k/x]E$, we must have by ($\mathtt{let}_P$) $\vdash_P\ \Gamma \cup \Gamma'\ \triangleright\ E_k\colon \sigma'$ and $\vdash_P\ \Gamma \cup \Gamma' \cup \{x_k\colon \sigma'\}\ \triangleright\ [x_k/x]E\colon \sigma$; by inductive hypothesis, take $\sigma' \equiv \sigma$. Since $\mathbf{fix}\, x.E$ is typable, then all the E_k are also typable. As such, each E_k must have a principal type $\mu_k \sqsubseteq \sigma$. ∎

Our goal is to now show that some successive E_k, E_{k+1} must have the same *principal* type.

Proposition 4.3 *For all types σ, $\mathcal{S}_\sigma = \{[\alpha]\colon\ \alpha \sqsubseteq \sigma\}$ is a finite set.*

Proof. Define the *length* of a monotype as $|t| = 1$, $|\tau_0 \to \tau_1| = |\tau_0| + |\tau_1|$. If $\alpha \sqsubseteq \sigma$, then $\langle\alpha\rangle \supseteq \langle\sigma\rangle$, hence $\overline{\sigma} \in \langle\alpha\rangle$. Since $|[\tau/t]\beta| \geq |\beta|$ (substitution cannot decrease length), we know $|\overline{\alpha}| \leq |\overline{\sigma}|$, and without loss of generality, the number of quantifiers preceding $\overline{\alpha}$ in α is bounded by $|\overline{\alpha}|$. ∎

Proposition 4.4 *Let $\vdash_P \Gamma\ \rhd\ E_i\colon \mu_i$ be a principal typing, where E is* **fix***-free. Then for all $i \geq 0$, $\mu_i \sqsubseteq \mu_{i+1}$.*

Proof. By induction on i. The basis is when $i = 0$: in this case, $\mu_0 \equiv \forall t.t \sqsubseteq \mu_1$.

For the inductive step, assume by inductive hypothesis that $\mu_i \sqsubseteq \mu_{i+1}$. Then $\vdash_P \Gamma \cup \{x\colon \mu_i\}\ \rhd\ E\colon \mu_{i+2}$, since we can take the proof $\vdash_P \Gamma \cup \{x\colon \mu_{i+1}\}\ \rhd\ E\colon \mu_{i+2}$, and note that any instantiation of $x\colon \mu_{i+1}$ to a monotype can also be carried out if $x\colon \mu_i$. Since $\vdash_P \Gamma\ \rhd\ E_i\colon \mu_i$, by ($\mathtt{let}_P$) it is clear that $\vdash_P \Gamma\ \rhd\ E_{i+1}\colon \mu_{i+2}$. By the principality of $\vdash_P \Gamma\ \rhd\ E_{i+1}\colon \mu_{i+1}$, we then know that $\mu_{i+1} \sqsubseteq \mu_{i+2}$. ∎

Lemma 4.5 *Let E be* **fix***-free. Then $\vdash_P \Gamma \cup \{x\colon \sigma\}\ \rhd\ E\colon \sigma$ if and only if there exists $k \geq 0$ and type $\sigma' \sqsubseteq \sigma$ where $\vdash_P \Gamma\ \rhd\ E_k\colon \sigma'$ and $\vdash_P \Gamma\ \rhd\ E_{k+1}\colon \sigma'$.*

Proof. Proposition 4.1 proves the 'if' direction. As for 'only if,' recall that $\mathcal{S}_\sigma = \{[\alpha]\colon\ \alpha \sqsubseteq \sigma\}$. By Proposition 4.2, $\mu_i \sqsubseteq \sigma$ for all $i \geq 0$, hence $[\mu_i] \in \mathcal{S}_\sigma$. As $\mathcal{S}_\sigma$ is finite by Proposition 4.3, there exists by the pigeonhole principle $0 \leq k < \ell \leq |\mathcal{S}_\sigma|$ where $[\mu_k] = [\mu_\ell]$. But by Proposition 4.4, $\mu_k \sqsubseteq \mu_{k+1} \sqsubseteq \mu_\ell$, hence $[\mu_k] = [\mu_{k+1}]$; take $\sigma' \equiv \mu_k$. ∎

Finally, we can state the equivalence theorem for the polytype and monotype inference systems with polymorphic fixpoint:

Theorem 4.6 *Let $\Gamma, \Gamma', \mathcal{F}$ be as in Definition 3.1. Then*

$$\vdash_P \Gamma \cup \Gamma'\ \rhd\ E'\colon \sigma \qquad \textit{if and only if} \qquad \vdash_M \Gamma\ \rhd\ [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]E'\colon \overline{\sigma}.$$

Proof. We augment the induction proof of Theorem 3.3 with the case $E' \equiv \mathbf{fix}\, x.E$. Without loss of generality, assume σ is a principal typing. If $\vdash_P \Gamma \cup \Gamma'\ \rhd\ \mathbf{fix}\, x.E\colon \sigma$, then by the argument of Lemma 3.2 $\vdash_P \Gamma \cup \Gamma' \cup \{x\colon \sigma\}\ \rhd\ E\colon \sigma$.

If E is **fix**-free, by the previous Theorem there exist $k \geq 0$ and $\sigma' \sqsubseteq \sigma$ such that $\vdash_P \Gamma \cup \Gamma'\ \rhd\ E_j\colon \overline{\sigma'}$ for $j \in \{k, k+1\}$. Since the E_j are **fix**-free, by Theorem 3.3 we have $\vdash_M \Gamma\ \rhd\ [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]E_j\colon \overline{\sigma'}$. Let $\tilde{E} \equiv [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]E$, so that $\vdash_M \Gamma\ \rhd\ \tilde{E}_j\colon \overline{\sigma'}$ for $j \in \{k, k+1\}$; by (fix_M) we have $\vdash_M \Gamma\ \rhd\ \mathbf{fix}\, x.\tilde{E}\colon \overline{\sigma'}$. However, note that $\mathbf{fix}\, x.\tilde{E} \equiv [F_1/y_1][F_2/y_2]\cdots[F_n/y_n]\mathbf{fix}\, x.E$.

In the case that E is not **fix**-free, we observe that using the inductive hypothesis, the above claims about **fix**-free expressions also hold with such stipulation. We then repeat the argument. ∎

5 Final remarks

It seems obvious that the polytype and monotype inference systems should be equivalent in their expressive power. When we use an expression defined with `let` and having a polytype, we instantiate the quantified type variables to be in accordance with the type context. Had we the code instead, we could type the code differently in each instance. In the ML module system, identifiers are bound to types without code, so that type inference can still take place; an obvious use for this facility is in incremental compilation. Of course, the module could instead give the code, but in practice the type is shorter. There are, however, examples where the type is much larger than the code, and these pathological examples provide the foundation for lower bounds on type inference [KM89, Mai90, KMM91]. In short: most general *specifications* (i.e., types) can be considerably longer than the programs implementing the specifications when the specification language is rich enough.

The equivalence proofs we have given are based on a fairly straightforward structural induction. The contribution of this paper, for the most part, is to give a precise definition of the equivalence. The lesson is simple: type polymorphism is not needed when you do not reuse code, and instead use separate copies of the same code. Our equivalence proofs explain a theory of type monomorphism in programming, where it becomes clear that the type polymorphism found in ML-like languages admits straightforward quantifier elimination procedures.

Acknowledgements. I would like to thank Gerd Hillebrand, Paris Kanellakis, and Lincoln Wallen for several stimulating and helpful discussions. In addition I would like to acknowledge the generous hospitality of the Computer Science Department at UC Santa Barbara, the Music Academy of the West, and the Cate School of Carpenteria during my stay in Santa Barbara in the summer of 1990, when I began work on this paper.

References

[Bar84] H. Barendregt. **The Lambda Calculus: Its Syntax and Semantics.** North-Holland, 1984.

[Car84] L. Cardelli. *Basic polymorphic type-checking.* **Science of Computer Programming**, 8(2), pp. 147–172, 1984.

[CF58] H. B. Curry and R. Feys. **Combinatory Logic I.** North-Holland, 1958.

[vanD79] D. van Daalen. **Logic and Structure.** Springer-Verlag, 1979.

[Dam85] L. Damas. *Type assignment in programming languages.* Ph. D. dissertation, CST-33-85, Computer Science Department, Edinburgh University, 1985.

[DM82] L. Damas and R. Milner. *Principal type schemes for functional programs.* In **9-th ACM Symposium on Principles of Programming Languages**, pp. 207–212, January 1982.

[Edi88] Edinburgh University. Postgraduate Examination Questions in Computation Theory, 1978–1988, ed. Donald Sannella. Laboratory for Foundations of Computer Science, Report ECS-LFCS-88-64.

[Gen69] G. Gentzen. **The Collected Papers of Gerhard Gentzen**, ed. E. Szabo. North-Holland, 1969.

[Gir72] J.-Y. Girard. *Interprétation Fonctionelle et Elimination des Coupures de l'Arithmetique d'Ordre Superieur.* Thèse de Doctorat d'Etat, Université de Paris VII, 1972.

[Han87] P. Hancock. *Polymorphic type-checking.* In **The Implementation of Functional Programming Languages**, by Simon Peyton-Jones. Prentice-Hall, 1987.

[HMT90] R. Harper, R. Milner, M. Tofte. **The Definition of Standard ML.** MIT Press, 1990.

[HM91] F. Henglein and H. Mairson. *The complexity of type inference for higher-order typed lambda calculi.* **Proceedings of the 18-th ACM Symposium on the Principles of Programming Languages**, January 1991, pp. 119–130.

[HS87] R. Hindley and J. Seldin. **Introduction to Combinators and Lambda Calculus.** Cambridge University Press, 1987.

[HW88] P. Hudak and P. L. Wadler, editors. *Report on the functional programming language Haskell.* Yale University Technical Report YALEU/DCS/RR656, 1988.

[KM89] P. C. Kanellakis and J. C. Mitchell. *Polymorphic unification and ML typing.* Brown University Technical Report CS-89-40, August 1989. Also in **Proceedings of the 16-th ACM Symposium on the Principles of Programming Languages**, pp. 105–115, January 1989.

[KMM91] P. C. Kanellakis, H. G. Mairson, and J. C. Mitchell. *Unification and ML type reconstruction.* In **Computational Logic: Essays in Honor of Alan Robinson**, ed. J.-L. Lassez and G. Plotkin. MIT Press, 1991.

[KTU90] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. *Undecidability of the semi-unification problem.* **Proceedings of the 22nd ACM Symposium on Theory of Computing**, May 1990. (See also Boston University Technical Report, October 1989).

[Mai90] H. G. Mairson. *Deciding ML typability is complete for deterministic exponential time.* In **Proceedings of the 17-th ACM Symposium on the Principles of Programming Languages**, pp. 382–401, January 1990.

[Mil78] R. Milner. *A theory of type polymorphism in programming.* **Journal of Computer and System Sciences** 17, pp. 348–375, 1978.

[Mit90] J. C. Mitchell. *Type systems for programming languages.* To appear as a chapter in the **Handbook of Theoretical Computer Science**, van Leeuwen et al., eds. North-Holland, 1990.

[MH88] J. C. Mitchell and R. Harper. *The essence of ML.* In **Proceedings of the 15-th ACM Symposium on Principles of Programming Languages**, pages 28–46, January 1988.

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

[Rey74] J. C. Reynolds. *Towards a theory of type structure.* In **Proceedings of the Paris Colloquium on Programming**, Lecture Notes in Computer Science 19, Springer Verlag, pp. 408–425, 1974.

[Rob65] J. A. Robinson. *A machine oriented logic based on the resolution principle.* **Journal of the ACM** 12(1):23–41, 1965.

[Tai67] W. W. Tait. *Intensinal interpretation of functionals of finite type I.* **J. Symbolic Logic** 32, pp. 198–212, 1967.

[Tur85] D. A. Turner. *Miranda: A non-strict functional language with polymorphic types.* In **IFIP International Conference on Functional Programming and Computer Architecture**, Nancy, Lecture Notes in Computer Science 201, pp. 1–16, Springer-Verlag, 1985.

[Wan87] M. Wand. *A simple algorithm and proof for type inference.* **Fundamenta Informaticae** 10 (1987).