Principal Typing and Mutual Recursion

Lucília Figueiredo[1] and Carlos Camarão[2]

[1] Universidade Federal de Ouro Preto, DECOM-ICEB
Ouro Preto 35400-000, Brasil
lucilia@dcc.ufmg.br
[2] Universidade Federal de Minas Gerais, DCC-ICEX
Belo Horizonte 31270-010, Brasil
camarao@dcc.ufmg.br

Abstract. As pointed out by Damas[Dam84], the Damas-Milner system (ML) has principal types, but not principal typings. Damas also defined in his thesis a slightly modified version of ML, that we call ML′, which, given a typing context and an expression, derives exactly the same types, and provided an algorithm (named as T) that infers principal typings for ML′. This work extends each of ML′ and T with a new rule for typing mutually recursive let-bindings. The proposed rule can type more expressions than the corresponding rule used in ML, by allowing mutually recursive definitions to be used polymorphically by other definitions.

1 Introduction

It is well known that the Damas-Milner type system [Mil78, DM82] has principal types, but not principal typings, as pointed out by Damas [Dam84] and stressed by Jim [Jim96]. The usefulness of the principal typing property has already been adressed in [Jim96], who suggests that principal typings are the key ingredient for efficiently solving the problem of type inference for mutually recursive definitions.

A slightly modified type system for core-ML that types exactly the same terms, and has principal typings, was defined by Damas [Dam84] in his thesis. This type system, which we call ML′, uses exactly the same syntax of types of the Damas-Milner system (ML), but allows a variable to be bound to several distinct types in a typing context. Damas also gave a type inference algorithm that infers principal typings for ML′, which he named algorithm T.

This work extends ML′ and T with a new rule for typing mutually recursive let-bindings, that allows mutually recursive definitions to be used polymorphically by other definitions.

The rest of this paper is organized as follows. Section 2 provides formal definitions of the concepts of principal type and principal typing that apply to any type system, by parameterizing the definitions by a suitable order on types. Section 3 defines a partial order on ML types. Section 4 presents type system ML′ and algorithm T_o (a slightly modified version of algorithm T), which infers principal typings for ML′. Section 5 extends ML′ and T_o for typing mutually recursive polymorphic definitions. Section 6 discusses related work and Section 7 presents our conclusions.

2 Principal Typing

The following syntactic meta-variables are used, ranging over the following sets of syntactic terms: x, y, for variables, e for expressions, σ for types, τ for simple types, α, β for type variables and Γ for typing contexts, a finite set of pairs $x : \sigma$.

If $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$, also written as $\{x_i : \sigma_i\}^{i=1..n}$, then $dom(\Gamma) = \{x_1, \ldots, x_n\}$. We use this superscript notation similarly for other sets. If Γ is a typing context in which each x occurs only once, then: if $x : \sigma \in \Gamma$, then σ is the type of x in Γ, denoted by $\Gamma(x)$; $\Gamma \ominus x$ is defined as $\Gamma - \{x : \Gamma(x)\}$; $(\Gamma, x : \sigma)$ is defined as $(\Gamma \ominus x) \cup \{x : \sigma\}$. Analogous definitions are assumed for the case where Γ may have more than one type assumption for the same variable.

A typing formula $\Gamma \vdash e : \sigma$ asserts that expression e has type σ under the assumptions given by typing context Γ.

The set of free variables of term e, denoted by $fv(e)$, and the set of free type variables of type σ (and of a typing context Γ), denoted by $tv(\sigma)$ ($tv(\Gamma)$), have the usual definitions.

Definition 1 (Typing Problem). A typing problem is a pair (e, Γ).

Note the possibility of including a typing context in a typing problem, that allows the use of fixed (predefined) assumptions to be considered in typing solutions, e.g. { `True: Bool, False: Bool, 1: Int, ...` }.

Definition 2 (Typing Solution). A solution to a typing problem (e, Γ_0) in a given type system is a pair (Γ, σ) such that $\Gamma \vdash e : \sigma$ is provable in this type system and, if $x \in fv(e) \cap dom(\Gamma_0)$, then $\Gamma(x) = \Gamma_0(x)$.

For example, given the typing problem $(f\ x, \{x : \alpha, f : \alpha \to \beta\})$, a typing solution in the type system of core-ML is $(\beta, \{x : \alpha, f : \alpha \to \beta\})$ (in this case this would be the only solution to this typing problem).

The definition of principal typing below is parameterized by a suitable partial order $\preceq$ on types of the relevant type system. This partial order induces a corresponding partial order $\preceq$ on typing contexts, representing "requirements on variables" occurring in these contexts, as well as an ordering on typings (i.e. pairs (Γ, σ)). The principal typing for a typing problem is thus defined simply as the $\preceq$-smallest element of the set of typings which are solutions to this problem.

Definition 3 (Ordering on typing contexts). Given a partial order $\preceq$ on types, a corresponding partial order $\preceq$ on typing contexts is defined by:

$$\Gamma \preceq \Gamma' = (x \in dom(\Gamma) \text{ implies that } x \in dom(\Gamma') \text{ and } \Gamma'(x) \preceq \Gamma(x))$$

From this definition, $\emptyset \preceq \Gamma$ is vacuously true, for any Γ. Informally, $\Gamma \preceq \Gamma'$ can be read as "Γ *requires less* (of its variables) *than* Γ'". Typing context Γ requires less than Γ' if for each assumption $x : \sigma$ in Γ there is an assuption $x : \sigma'$ in Γ' such that $\sigma' \preceq \sigma$ (i.e. σ' *provides more* than σ). This definition also applies to typing contexts which allow more than one typing for the same variable, provided that a partial ordering on sets of types is defined:

Definition 4 (Ordering on sets of types). Given partial order $\preceq$ on types, the partial order $\preceq$ on sets of types is defined by $\{\sigma_i\}^{i=1..n} \preceq \{\sigma'_j\}^{j=1..m}$ if, for each σ'_j, $j = 1..m$, there exists σ_i, $1 \leq i \leq n$, such that $\sigma_i \preceq \sigma'_j$.

Definition 5 (Ordering on typings). Given a typing problem (e, Γ_0), for any typing solutions (Γ, σ) and (Γ', σ') to this typing problem, we define:

$$(\Gamma, \sigma) \preceq (\Gamma', \sigma') = \big(\Gamma \preceq \Gamma' \text{ and } (\Gamma = \Gamma' \text{ implies } \sigma \preceq \sigma')\big)$$

Definition 6 (Principal Typing). The *principal typing* solution to a typing problem (e, Γ) is the $\preceq$-smallest element of the set of all solutions to this typing problem, if it exists; otherwise there is no principal typing for (e, Γ).

Definition 7 (Principal Type). Given a typing problem (e, Γ_0), σ is the *principal type* for expression e in context Γ_0, if there exists a principal typing solution (σ, Γ) to this typing problem; otherwise, there is no principal type for e in Γ_0.

According to Definition 2, the principal typing for problem (e, Γ_0) is a solution (σ, Γ) such that Γ requires less of its variables than any other solution and, for this minimal context, the principal typing solution gives the smallest type.

3 Parametric Polymorphism

The context-free syntax of types and terms of `ML` [Mil78, DM82] is given below (meta-variables x and α range, resp., over a countably infinite set of *variables* and a countably infinite set of *type variables*): [1]

$$\begin{array}{lll} \text{Simple Types} & \tau & ::= \alpha \mid \tau \to \tau' \\ \text{Types} & \sigma & ::= \forall\alpha.\, \sigma \mid \tau \\ \text{Expressions} & e & ::= x \mid \lambda x.\, e \mid e\, e' \mid \texttt{let}\ x = e\ \texttt{in}\ e' \end{array}$$

An equivalent version of `ML` type system [Hen93, KTU93, KTU94] is presented in Fig.1.

Predicate $inst(\forall(\alpha_i)^{i=1..n}.\tau', \tau)$ holds when $\tau = \tau'[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$, for some $\tau_1, \ldots, \tau_n$. We use $\forall(\alpha_i)^{i=1..n}.\tau$ as abbreviation for $\forall\alpha_1.\cdots\forall\alpha_n.\tau$, where $n \geq 0$. We sometimes drop the superscripts ($i = 1..n$), and write $\forall\alpha_i.\tau$.

A solution for typing problem (e, Γ_0) in the `ML` type system is a typing (Γ, σ) such that $\Gamma \vdash e : \tau$ is provable by the given rules and $close(\tau, \sigma, tv(\Gamma))$. Predicate *close* is defined, for any type σ and any set of type variables V, by $close(\sigma, \sigma', V) = (\sigma' = \forall(\alpha_i)^{i=1..n}.\sigma)$, where $\{\alpha_i\}^{i=1..n} = tv(\sigma) - V$. [2] We overload *close* to define $close(\sigma, \sigma') = close(\sigma, \sigma', \emptyset)$.

[1] As usual (c.f. [MH93, KTU93, KW94]), we do not include term constants, neither type constants (constructors) other than the functional type constructor, for simplicity of notation. The results in this work are also valid if they are included.

[2] The reader should not be confused by the overloading of '='. The first occurrence in the definition of *close* means 'is defined by' while the second represents a predicate.

$$\Gamma, x : \sigma \vdash x : \tau \quad \text{where } inst(\sigma, \tau) \qquad \texttt{(VAR)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \sigma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \tau_2} \; close(\tau_1, \sigma, tv(\Gamma)) \qquad \texttt{(LET)}$$

$$\frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x.\, e : \tau' \to \tau} \qquad \texttt{(ABS)}$$

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\, e_2 : \tau} \qquad \texttt{(APPL)}$$

Figure 1: Type System ML

The ordering on types for languages with quantified and simple types formed by means of type variables and type constructors should consider relations between quantified types and relations on types obtained by substitutions (which are functions from type variables to types).

Meta-variable S is used to range over substitutions, which are functions from type variables to types; $S\sigma$ represents the capture-free operation of substituting all free occurrences of type variables α in σ by $S(\alpha)$; $S\Gamma$ represents the typing context obtained by replacing each $x : \sigma \in \Gamma$ with $x : S\sigma$. We define $S \dagger \{\alpha \mapsto \tau\}(\beta) = S(\beta)$, if $\beta \neq \alpha$, and τ if $\beta = \alpha$; and $\sigma[\tau/\alpha] = (id \dagger \{\alpha \mapsto \tau\})\sigma$, where id is the identity substitution.

We define below a partial order $\preceq$, on quantified types σ and simple types τ, formed by means of type variables and type constructors, called *ordering induced by parametric polymorphism*. A more complex definition would be necessary in order to consider relations on types with quantifications not restricted to occur at the outermost level.

The definition of the ordering on types is simplified by observing the convention that types are syntatically equal modulo renaming of bound type variables, reordering of adjacent quantifiers and elimination of unnecessary quantifiers.

Definition 8 (Parametric Polymorphism). Partial order $\preceq$ on ML types is defined by $\sigma \preceq \sigma'$ if $\sigma \preceq_S \sigma'$, for some substitution S, where the relation $\sigma \preceq_S \sigma'$ is defined as follows:

1. $\sigma \preceq_S \sigma'$, if $S\sigma' = \sigma$
2. $\forall\alpha.\sigma \preceq_{id} \sigma[\tau/\alpha]$, for any τ, and
   $\sigma \preceq_{id} \forall\alpha.\sigma'$ if $\sigma \preceq_{id} \sigma'$ and $\alpha \notin tv(\sigma)$
3. $\sigma \preceq_S \sigma'$, if $\sigma \preceq_{S_1} \sigma_1$, $\sigma_1 \preceq_{S_2} \sigma'$ and $S = S_2 \circ S_1$

Note that relation $inst(\sigma, \tau)$ can be expressed as $\sigma \preceq_{id} \tau$.

We have, for example: $\forall\alpha.\, \beta \to \alpha \preceq_{id} \forall\alpha_1, \alpha_2.\, \beta \to (\alpha_1 \to \alpha_2)$, from (*2*), $\forall\alpha_1, \alpha_2.\, \beta \to (\alpha_1 \to \alpha_2) \preceq_{id} \beta \to (\alpha_1 \to \alpha_2)$, from (*2*), and $\beta \to (\alpha_1 \to \alpha_2) \preceq_S \alpha$, from (*1*), for any S that maps α to $\beta \to (\alpha_1 \to \alpha_2)$.

We comment briefly on some simple properties of the ordering on types that come directly from the given definitions. The first one is the antimonotonicity

of quantification over an ordering that considers only rules *1* and *3*: if $S\sigma_1 = \sigma_2$ then $\sigma'_1 \preceq_{id} \sigma'_2$, where $close(\sigma_1, \sigma'_1)$ and $close(\sigma_2, \sigma'_2)$. This reflects simply that if "less is required" of the type of a given expression e (in the sense that substitutions on free type variables are not "required" for the expression to be well-typed) then the type of the term obtained by closing e (i.e. by introducing λ-abstractions over all its free variables) "provides more". For example, the type of $\lambda x.\, x$ provides more (is more general) than, say, $\lambda x.\, \lambda f.\, f\, x$, since less is required of the variable x (in expression x than in $\lambda f.\, f\, x$).

Another, rather trivial property, is that the function type constructor is monotonic, in both arguments, over an ordering that considers only rules *1* and *3*: If $S\sigma_1 = \sigma'_1$ and $S\sigma_2 = \sigma'_2$, then $S(\sigma_1 \to \sigma_2) = \sigma'_1 \to \sigma'_2$. It follows that the function type constructor is neither monotonic nor antimonotonic, with respect to $\preceq$ (either in the first or the second argument).

4 Principal typings for ML

The fact that ML type system has principal types, not principal typings, can be seen by considering that each of the following infinite list of typing contexts can be used to derive a type for $x\, x$, and for each such typing context the next one in the list is smaller: $\{x\colon \forall\alpha.\, \alpha\}, \{x\colon \forall\alpha.\, \alpha\to\alpha\}, \{x\colon \forall\alpha.\, (\alpha\to\alpha)\to(\alpha\to\alpha)\}, \ldots$

As pointed out by Jim [Jim96], in a system that lacks principal typings, one can still achieve the benefits of this property by finding a suitable "representation" for all its possible typings, relaxing the principal typing condition that the representatives themselves be typings in this system: the representation may be, for example, a typing in another system.

Type system ML′ is a slightly modified version of a type system defined by Damas [Dam84], which is also similar to ML type system. A typing context in ML′ allows multiple assumptions for the same variable. We distinguish between λ-bound variables and let-bound variables and use meta variable u to denote a λ-bound variable and meta-variable x to denote either a λ-bound or a let-bound variable, when the distinction is clear from the context or is not important.

We define $\Gamma^{\mathrm{u}} = \{x : \sigma \in \Gamma \mid x \text{ is a } \lambda\text{-bound variable }\}$ and $tv_{\mathrm{u}}(\Gamma) = tv(\Gamma^{\mathrm{u}})$. We also define $\Gamma|_X = \{x : \sigma \in \Gamma \mid x \in X\}$, for any set of (term) variables X, and we use $\#V$ for the cardinality of set V.

With the distinction between λ-bound and let-bound variables, a typing context Γ is valid in ML if it satisfies the following conditions: i) $x \in dom(\Gamma)$ implies that $\Gamma(x) = \{\sigma\}$, for some type σ, and if x is a λ-bound variable then σ is a simple type; ii) $tv(\Gamma) = tv_{\mathrm{u}}(\Gamma)$.

ML′ differs from ML only in rule (VAR), which is substituted by the rule:

$$\Gamma \cup \{x : \sigma\} \vdash x : \tau \quad \text{where } \sigma \preceq_{id} \tau \qquad (\mathtt{VAR}')$$

and in rule (LET), where the side condition $close(\tau_1, \sigma, tv(\Gamma))$ is substituted by $close(\tau_1, \sigma, tv_{\mathrm{u}}(\Gamma))$.

The relationship between typability in ML and typability in ML′ is formally stated by theorems 1 and 2 at the end of this section.

$$\Gamma_0 \vdash x : (\tau, \Gamma) \quad \text{where } (\tau, \Gamma) = \begin{cases} (lcg(\Gamma_0(x)), \Gamma_0) & \text{if } \#\Gamma_0(x) \geq 1 \\ (\alpha, \Gamma_0 \cup \{x : \alpha\}) & \text{otherwise, where } \alpha \text{ is fresh} \end{cases} \qquad (\texttt{VAR}^{\circ})$$

$$\frac{\Gamma_0 \vdash e_1 : (\tau_1, \Gamma_1) \quad \Gamma_0', x : \sigma \vdash e_2 : (\tau_2, \Gamma_2)}{\Gamma_0 \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : (S\tau_2, S\Gamma_1 \cup (S\Gamma_2 \ominus x))} \quad \begin{array}{l} \Gamma_0' = \Gamma_1|_{dom(\Gamma_0)} \cup \Gamma_1^{\mathrm{u}} \\ close(\tau_1, \sigma, tv_{\mathrm{u}}(\Gamma_1)) \\ S = unify(\mathcal{E}_{\mathrm{u}}(\Gamma_1, \Gamma_2)) \end{array} \qquad (\texttt{LET}^{\circ})$$

$$\frac{\Gamma_0, u : \alpha \vdash e : (\tau, \Gamma)}{\Gamma_0 \vdash \lambda u.\, e : (\tau' \to \tau, \Gamma \ominus u)} \quad \begin{array}{l} \{\tau'\} = \Gamma(u) \\ \alpha \text{ is a fresh type variable} \end{array} \qquad (\texttt{ABS}^{\circ})$$

$$\frac{\Gamma_0 \vdash e_1 : (\tau_1, \Gamma_1) \quad \Gamma_0 \vdash e_2 : (\tau_2, \Gamma_2)}{\Gamma_0 \vdash e_1\, e_2 : (S\alpha, S\Gamma_1 \cup S\Gamma_2)} \quad \begin{array}{l} S = unify(\mathcal{E}_{\mathrm{u}}(\Gamma_1, \Gamma_2) \cup \{\tau_1 = \tau_2 \to \alpha\}) \\ \alpha \text{ is a fresh type variable} \end{array} \qquad (\texttt{APPL}^{\circ})$$

Figure 2: Type System $\mathtt{T}_{\mathrm{o}}$

Algorithm $\mathtt{T}_{\mathrm{o}}$, defined in Fig.2, infers principal typings for $\mathtt{ML}'$. $\mathtt{T}_{\mathrm{o}}$ is essentially equivalent to algorithm T defined by Damas, except that it is modified to have a typing context as input, in accordance to our definition of a typing problem.

As with $\mathtt{ML}'$, $\mathtt{T}_{\mathrm{o}}$ allows typing contexts to have more than one assumption for the same variable. For example, the principal typing solution to typing problem $(x\, x, \emptyset)$ is $(\alpha', \{x : \alpha, x : \alpha \to \alpha'\})$, where α, α' are fresh type variables.

We use $lcg(\{\tau_i\}^{i=1..n})$ to denote the least common generalisation of the set of types $\{\tau_i\}^{i=1..n}$. A simplification is used (as lcg is not really a function), that considers lcg as a function by choosing any (representative of the equivalence class of types) τ that is a least common generalisation of $\{\tau_i\}^{i=1..n}$ (where $\tau \equiv \tau'$ if they are equal except for renaming of fresh type variables), and we assume that $lcg(\{\tau\}) = \tau$. We overload lcg and define:

$$lcg(\{\sigma_i\}) = \tau \text{ where } \begin{array}{l} \sigma_i = \forall(\alpha_{ij})^{j=1..n_i}.\tau_i, \text{ for } i = 1..n \\ \tau = lcg(\{\tau_i[\alpha'_{ij}/\alpha_{ij}]^{j=1..n_i}\}^{i=1..n}) \\ \alpha'_{ij} \text{ is a fresh type variable for } j = 1..n_i,\ i = 1..n \end{array}$$

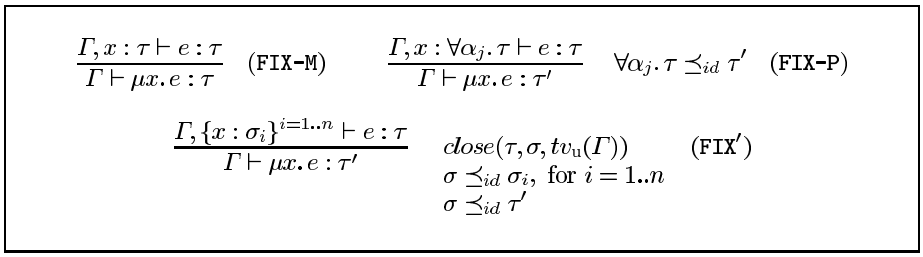
$$lcg(\Gamma) = \{x : \sigma \mid\ x \in dom(\Gamma) \text{ and } close(lcg(\Gamma(x)), \sigma, tv_{\mathrm{u}}(\Gamma))\}$$

Function unify, used in rules $(\texttt{LET}^{\circ})$ and $(\texttt{APPL}^{\circ})$, computes the most general unifier of a set of type equations. $\mathcal{E}_{\mathrm{u}}(\Gamma, \Gamma')$ represents the set of equations on the types of each λ-bound variable occurring both in Γ and Γ', that is:

$$\mathcal{E}_{\mathrm{u}}(\Gamma, \Gamma') = \{\tau = \tau' \mid u : \tau \in \Gamma, u : \tau' \in \Gamma'\}$$

The relationship between ML and $\mathtt{ML}'$ is formally stated by theorems 1 and 2 below, where we use $\vdash$ and $\vdash'$ for derivations in ML and $\mathtt{ML}'$, respectively. A typing context Γ in $\mathtt{ML}'$ must satify the condition that $lcg(\Gamma)$ is a typing context in ML.

Theorem 1. Let Γ be an ML typing context. If $\Gamma \vdash e : \tau$ is provable then $\Gamma \vdash' e : \tau$ is provable.



$$\frac{\Gamma, x:\tau \vdash e:\tau}{\Gamma \vdash \mu x.e:\tau} \quad \texttt{(FIX-M)} \qquad \frac{\Gamma, x:\forall\alpha_j.\tau \vdash e:\tau}{\Gamma \vdash \mu x.e:\tau'} \quad \forall\alpha_j.\tau \preceq_{id} \tau' \quad \texttt{(FIX-P)}$$

$$\frac{\Gamma, \{x:\sigma_i\}^{i=1..n} \vdash e:\tau}{\Gamma \vdash \mu x.e:\tau'} \quad \begin{array}{l} close(\tau, \sigma, tv_{\mathrm{u}}(\Gamma)) \\ \sigma \preceq_{id} \sigma_i, \text{ for } i = 1..n \\ \sigma \preceq_{id} \tau' \end{array} \quad \texttt{(FIX')}$$

Figure 3: Typing rules for polymorphic recursion

Theorem 2. Let Γ be an ML′ typing context. If $\Gamma \vdash' e : \tau$ is provable then $lcg(\Gamma) \vdash e : \tau$ is provable.

Corollary 1. If Γ is an ML typing context, then $\Gamma \vdash e : \tau$ is provable if and only if $\Gamma \vdash' e : \tau$ is provable.

The relationship between ML′ and $\mathtt{T}_{\mathrm{o}}$ is formally stated by theorems 3 and 4 below, where we use $\vdash'$ and $\vdash^{\circ}$ for derivations in ML′ and $\mathtt{T}_{\mathrm{o}}$, respectively.

Theorem 3 (Soundness $\mathtt{T}_{\mathrm{o}}$-ML′). Let Γ_0 be an ML typing context. If $\Gamma_0 \vdash^{\circ} e : (\tau, \Gamma)$ is provable, then Γ is an ML′ typing context and $\Gamma \vdash' e : \tau$ is provable.

Theorem 4 (Principal Typing $\mathtt{T}_{\mathrm{o}}$-ML′). Let (e, Γ_0) be a typing problem such that $\Gamma_0|_{fv(e)}$ is an ML typing context. If $\Gamma_0 \vdash^{\circ} e : (\tau, \Gamma)$ is provable, then (σ_p, Γ_p) is the principal typing solution for (e, Γ_0) in ML′, where $\Gamma_p = \Gamma|_{fv(e)}$ and $close(\tau, \sigma_p, tv_{\mathrm{u}}(\Gamma_p))$; otherwise, (e, Γ_0) has no solution in ML.

5 Mutual Recursion

The language of core-ML is extended with polymorphic recursive definitions by including expressions of the form $\mu x.\, e$, which represent expression `let` $x = e$ `in` x, where e may contain occurrences of x (more often written `letrec` $x = e$ `in` x).

Languages that restrict polymorphism and use a decidable type inference algorithm are based on an extension of ML with rule (FIX-M), presented in Figure 3. Note that this rule only allows the defined variable (x) to be used *monomorphicaly* in the body of its definition (e).

In an attempt to overcome this limitation, Mycroft [Myc84] and Meertens [Mee93] have independently proposed rule (FIX-P), also presented in Figure 3.

Expression $\mu x.\, xx$ is a simple example of an expression that is typable under rule (FIX-P), but is not typable under rule (FIX-M). For a more useful example, consider the following definition:

```
data Seq a = Nil | Cons a (Seq (a,a))

length Nil          = 0
length (Cons x s) = 1 + 2 * (length s)
```

$$\frac{\text{for } j{=}1..n \quad \Gamma, \{x_i : \tau_i\}^{i=1..n} \vdash e_j : \tau_j \qquad \Gamma, \{x_i : \sigma_i\}^{i=1..n} \vdash e : \tau}{\Gamma \vdash \texttt{letrec } \{x_i = e_i\}^{i=1..n} \texttt{ in } e : \tau} \quad close(\tau_i, \sigma_i, tv(\Gamma)),\ i{=}1..n \qquad (\texttt{LETREC-M})$$

$$\frac{\text{for } j{=}1..n \quad \Gamma, \{x_i : \sigma_i\}^{i=1..n} \vdash e_j : \tau_j \qquad \Gamma, \{x_i : \sigma_i\}^{i=1..n} \vdash e : \tau}{\Gamma \vdash \texttt{letrec } \{x_i = e_i\}^{i=1..n} \texttt{ in } e : \tau} \quad close(\tau_i, \sigma_i, tv(\Gamma)),\ i = 1..n \qquad (\texttt{LETREC-P})$$

$$\frac{\text{for } j{=}1..n \quad \Gamma, \{x_i : \sigma_{i_1}, \ldots, x_i : \sigma_{i_{n_i}}\}^{i=1..n} \vdash e_j : \tau_j \qquad \Gamma, \{x_i : \sigma_i\}^{i=1..n} \vdash e : \tau}{\Gamma \vdash \texttt{letrec } \{x_i = e_i\}^{i=1..n} \texttt{ in } e : \tau} \quad \begin{array}{l} close(\tau_i, \sigma_i, tv_{\mathrm{u}}(\Gamma)),\ i{=}1..n \\ \sigma_i \preceq_{id} \sigma_{i_j},\ j{=}1..n_i,\ i{=}1..n \end{array} \qquad (\texttt{LETREC}')$$

Figure 4: Typing rules for mutually recursive definitions

Data type *Seq a* represents sequences of $2^k - 1$ elements, $k = 0, 1, \ldots$. The data type is non-uniform because the recursive component *Seq* (a, a) is different from *Seq a*. Function *length* computes the length of sequences in time $O(log\ n)$. Its definition uses polymorphic recursion, since *length* receives a value of type *Seq a*, for any a, and returns an integer, but calls itself with type *Seq* $(a, a) \to$ `int`.

Type system ML′ is extended with rule (FIX′), also presented in Figure 3, for typing polymorphic recursion. Note that, as in rule (FIX-P), the recursive variable (x) can be used polymorphicaly in the body of its definition (e).

The idea behind rule (FIX′) is the same as the one behind the following rule (FIX), also suggested by Mycrof [Myc84], namely, that each of the finite occurrences of x in e may have a different (simple) type (so long as they can be used to type e and are instances of the derived type for e):

$$\frac{\Gamma \vdash \lambda x_1 \ldots x_n.e' : \tau_1 \to \ldots \tau_n \to \tau}{\Gamma \vdash \mu x.e : \tau'} \qquad (\texttt{FIX})$$

where e is an expression with n occurrences of x, e' is e with each occurrence of x renamed to a fresh variable x_i, $\tau_1, \ldots, \tau_n$ are simple types, $\sigma \preceq_{id} \tau_i$ (or, equivalently, $inst(\sigma, \tau_i)$), for $i = 1..n$, and $\sigma \preceq_{id} \tau'$, where $close(\tau, \sigma, tv(\Gamma))$.

It is also interesting to extend the language for expressing (possibly) mutually recursive definitions, by adding the construct `letrec` $x_1 = e_1, \ldots, x_n = e_n$ `in` e, which we also write as `letrec` $\{x_i = e_i\}^{i=1..n}$ `in` e, where all x_i are distinct.

Corresponding rules (LETREC-M), (LETREC-P) and (LETREC′) are given in Figure 5. Rule (LETREC-M) allows the defined variables to be used only monomorphicaly in the body of the mutually recursive definitions. For example, the following definitions cannot be typed with rule (LETREC-M):

```
map f xs        = [ f x | x <- xs ]
complementList  = map not
squareList      = map square
```

This program is not typeable under rule (LETREC-M), when presented as a single, mutual recursive definition, since function *map* is used polymorphically by the other functions, and the rule requires these functions to be typed under the assumption that *map* has a simple type. Note that *map* does not depend on the other functions and the program could be typed by the rule above if *map* is placed in a separate recursive definition.

This strategy is in fact used to type any unordered set of definitions. The call graph of the program is examined to determine a set of strongly connected components $B_1, \ldots, B_m$ of mutually recursive bindings, and the $B_i s$ are topologically sorted to determine an order in which to type the definitions. That is, to check that a program $x_1 = e_1, \ldots, x_n = e_n$ is well typed, one performs type inference on the expression `letrec` B_1 `in` $\ldots$(`letrec` B_m `in` 0) derived from the call graph of the program, where each B_i is a strongly connected component of mutually recursive bindings and the B_is are topologically sorted.

The relation between typability in ML′ extended with rules (FIX′) and (LETREC′) and typability in ML extended with rules (FIX-P) and (LETREC-P), respectively, is formally stated by the theorems below.

Theorem 5. Let Γ be an ML typing context. If $\Gamma \vdash e : \tau$ is provable in ML+(FIX-P)+(LETREC-P), then $\Gamma \vdash' e : \tau$ is provable in ML′+(FIX′)+(LETREC′).

Theorem 6. Let Γ be an ML′ typing context. If $\Gamma \vdash' e : \tau$ is provable in ML′+(FIX′)+(LETREC′), then $lcg(\Gamma) \vdash e : \tau$ is provable in ML+(FIX-P)+(LETREC-P).

Type inference for the extension of ML with rule (FIX-P) has been proved to be undecidable, independently by Heiglein [Hen93] and by Kfoury et al. [KTU93]. On the other hand, type inference for ML′+(FIX′) is decidable, as we show below. We comment more on this apparent contradiction in Section 6, after presenting a type inference rule corresponding to rule (FIX′), and some simple examples.

Type inference algorithm $\mathtt{T}_o$ is extended with rule (FIX°), given in Figure 5, for inferring types of expressions of the form $\mu x.\, e$.

Notation $\mathcal{E}_p(\tau, \{\tau_i\}^{i=1..n}, V)$, used in this rule, represents the set of type equations defined as follows:

$$\mathcal{E}_p(\tau, \{\tau_i\}^{i=1..n}, V) = \{\tau_i' = \tau_i \mid \tau_i' = \tau[\alpha_{i_j}/\alpha_j]^{j=1..m}\}^{i=1..n}$$
$$\text{where } \{\alpha_j\}^{j=1..m} = tv(\tau) - V$$
$$\{\alpha_{i_j}\}^{i=1..n, j=1..m}, \text{ are fresh type variables}$$

As a first example of type inference by rule (FIX°) consider type inference for expression $\mu x.\, x\, x$, given an empty typing context. We have that $\emptyset \vdash^\circ x\, x : (\beta, \{x : \alpha \to \beta, x : \alpha\})$ is provable, where α, β are fresh type variables. According to rule (FIX°), we have that $\ell = 1$ and substitution S_1 is obtained as

$$S_1 = unify(\{\beta' = \alpha \to \beta,\ \beta'' = \alpha\})$$

Thus $S_1\Gamma = \Gamma = \{x\colon \alpha \to \beta,\ x\colon \alpha\}$, $\sigma = \forall\beta.\beta$, and the type inferred for $\mu x.\, x\, x$ is β (where β is implicitly quantified), since $\sigma \preceq_{id} \alpha \to \beta$, and $\sigma \preceq_{id} \alpha$.

$$\frac{\Gamma_0 \ominus x \vdash e : (\tau, \Gamma)}{\Gamma_0 \vdash \mu x.\, e : (S_\ell \tau, S_\ell(\Gamma \ominus x))} \quad (\texttt{FIX}^\circ)$$

where $\ell = max(1, \#(tv(\tau) - tv_{\mathrm{u}}(\Gamma)))$, $S_0 = id$
$S_i = unify(\mathcal{E}_{\mathrm{p}}(S_{i-1}\tau, S_{i-1}\Gamma(x), tv_{\mathrm{u}}(S_{i-1}\Gamma))) \circ S_{i-1}$, for $i = 1, \ldots, \ell$
$close(S_\ell \tau, \sigma, tv_{\mathrm{u}}(S_\ell \Gamma))$
$\sigma \preceq_{id} \tau'$, for each $\tau' \in S_\ell \Gamma(x)$

Figure 5: Type inference rule for recursive expressions

As another illustrative example, consider type inference for the following expression ($\mu h.\, e_h$), in an empty typing context:

$$\mu h.\, \lambda x.\, \lambda y.\, \texttt{if}\ h\ x\ y\ \texttt{==}\ y\ \texttt{then}\ (h\ y\ x)\ \texttt{+}\ 1$$

We have that $\emptyset \vdash e_h : (\alpha \to \beta \to \texttt{int}, \{h : \alpha \to \beta \to \beta, h : \beta \to \alpha \to \texttt{int}\})$ is provable. By rule ($\texttt{FIX}^\circ$), we have that $\ell = 2$, and substitution S_2 is obtained by the following sequence of unifications:

$$S_1 = unify\left(\left\{\begin{array}{l} \alpha' \to \beta' \to \texttt{int} = \alpha \to \beta \to \beta, \\ \alpha'' \to \beta'' \to \texttt{int} = \beta \to \alpha \to \texttt{int} \end{array}\right\}\right) \circ id$$

$$S_2 = unify\left(\left\{\begin{array}{l} \alpha' \to \texttt{int} \to \texttt{int} = \alpha \to \texttt{int} \to \texttt{int}, \\ \alpha'' \to \texttt{int} \to \texttt{int} = \texttt{int} \to \alpha \to \texttt{int} \end{array}\right\}\right) \circ S_1$$

Then $S_2\Gamma = \{h : \texttt{int} \to \texttt{int} \to \texttt{int}\}$ and the inferred type is $\texttt{int} \to \texttt{int} \to \texttt{int}$.

Expression $\mu f.\, \lambda x.f$ is a simple example of a recursive expression that is not typable. We have that $\emptyset \vdash^\circ \lambda x.f : (\alpha \to \beta, \{f : \beta\})$ is provable. By rule ($\texttt{FIX}^\circ$), we have that $\ell = 2$ and S_2 is obtained by the following sequence of unifications:

$$S_1 = unify(\{\alpha' \to \beta' = \beta\}) = id \dagger \{\beta \mapsto (\alpha' \to \beta')\} \circ id$$

$$S_2 = unify(\{\alpha'' \to \alpha' \to \beta' = \alpha' \to \beta'\}) \circ S_1$$

Then $S_2\Gamma = \{f : \alpha' \to \alpha' \to \beta'\}$ and $\sigma_f = \forall\alpha.\forall\alpha'.\forall\beta'.\, \alpha \to (\alpha' \to \alpha' \to \beta')$. Type inference fails since it does not hold that $\sigma_f \preceq_{id} \tau'$ for each $\tau' \in S_2\Gamma(f)$.

As an example of type inference for polymorphic recursion inside a λ-abstraction, consider type inference for $\lambda x.\, \mu f.\, x\ f$, given an empty typing context. We infer:

$$\emptyset \vdash^\circ x\ f : (\beta, \{x : \alpha \to \beta, f : \alpha\})$$

By rule ($\texttt{FIX}^\circ$), we have that $\ell = 1$ and substitution S_1 is obtained as:

$$S_1 = unify(\{\beta = \alpha\})$$

Then we have that $\emptyset \vdash \mu f.\, x\ f : (\alpha, \{x : \alpha \to \alpha\})$ is provable and, using rule ($\texttt{ABS}^\circ$), we have that $\emptyset \vdash \lambda x.\, \mu f.\, x\ f : (\alpha, \emptyset)$ is provable.

$$\frac{\begin{array}{r}\text{for } j{=}1..n \quad \Gamma_0 \ominus \{x_i\}^{i=1..n} \vdash e_j : (\tau_j, \Gamma_j) \\ \Gamma_0', \{x_i : \sigma_i\}^{i=1..n} \vdash e : (\tau, \Gamma)\end{array}}{\Gamma_0 \vdash \texttt{letrec } \{x_i = e_i\}^{i=1..n} \texttt{ in } e : (S\tau, \Gamma')} \quad (\texttt{LETREC}^{\circ})$$

where $\Gamma_\Diamond = \bigcup_i \Gamma_i$

$\ell = max(1, \#(tv(\{\tau_i\}^{i=1..n}) - tv_{\mathrm{u}}(\Gamma_\Diamond)))$, $S_0 = id$

$S_j = unify\left(\bigcup_i \mathcal{E}_{\mathrm{p}}(S_{j-1}\tau_i, S_{j-1}\Gamma_\Diamond(x_i), tv_{\mathrm{u}}(S_{j-1}\Gamma_i))\right) \circ S_{j-1}$, $j = 1, \ldots, \ell$

$S' = unify\left(\bigcup_{i=1..(n-1)} \mathcal{E}_{\mathrm{u}}(S_\ell\Gamma_i, S_\ell\Gamma_{(i+1)})\right) \circ S_\ell$

$\Gamma_0' = S'\Gamma_\Diamond|_{dom(\Gamma_0)} \cup \Gamma_\Diamond^{\mathrm{u}}$

$close(S'\tau_i, \sigma_i, tv_{\mathrm{u}}(S'\Gamma_i))$, for $i = 1, \ldots, n$

$\sigma_i \preceq_{id} \tau'$, for each $\tau' \in S'\Gamma_\Diamond(x_i)$, for $i = 1, \ldots, n$

$S = unify(\mathcal{E}_{\mathrm{u}}(S'\Gamma_\Diamond, \Gamma))$

$\Gamma' = (S\Gamma \cup SS'\Gamma_\Diamond) \ominus \{x_i\}^{i=1..n}$

Figure 6: Type inference rule for mutually recursive definitions

Our last example illustrattes type inference for nested mutual recursion. Consider type inference for the following expression, given an empty typing context:

$$\mu g.\, \mu f.\, \lambda u.\, \texttt{if } \mathit{true} \texttt{ then } g(u) \texttt{ else } f(3)$$

According to the type inference rules, we infer

$$\emptyset \vdash^{\circ} \lambda u.\, \texttt{if } \mathit{true} \texttt{ then } g(u) \texttt{ else } f(3) : (\alpha \to \beta, \{f : \texttt{int} \to \beta, g : \alpha \to \beta\})$$

By rule ($\texttt{FIX}^{\circ}$) we have that $\ell = 2$ and $S_2 = id \circ S_1$, where S_1 is obtained as:

$$S_1 = unify(\{\alpha_1 \to \beta_1 = \texttt{int} \to \beta\}) \circ id$$

Then we find that $\forall\alpha.\forall\beta.\, \alpha \to \beta$ has $\texttt{int} \to \beta$ as an instance, obtaining that typing $(\alpha \to \beta, \{g : \alpha \to \beta\})$ is inferred for $\mu f.\, \lambda u.\, \texttt{if } \mathit{true} \texttt{ then } g(u) \texttt{ else } f(3)$. After the unification and generalization steps, we find that $\alpha \to \beta$ is an instance of $\forall\alpha.\forall\beta.\, \alpha \to \beta$, inferring the validity of typing formula

$$\emptyset \vdash \mu g.\, \mu f.\, \lambda u.\, \texttt{if } \mathit{true} \texttt{ then } g(u) \texttt{ else } f(3) : (\alpha \to \beta, \emptyset)$$

Algorithm $\mathtt{T}_{\mathrm{o}}$ is extended with rule ($\texttt{LETREC}^{\circ}$), given in Figure 6, for inferring types of mutually recursive definitions. Rule ($\texttt{LETREC}^{\circ}$) can be obtained from rules ($\texttt{FIX}^{\circ}$) and ($\texttt{LET}^{\circ}$), by noting that expression $\texttt{letrec } \{x_i = e_i\}^{i=1..n} \texttt{ in } e$ can be rewritten as $\texttt{let } x = \mu x.\, (e_1', \ldots, e_n') \texttt{ in } e'$, where $e_i' = e_i[\pi_i\, x/x_i]^{i=1..n}$, for $i = 1, \ldots, n$, $e' = e[\pi_i\, x/x_i]^{i=1..n}$, $n \geq 1$, and $\pi_i\, (e_1, \ldots, e_n) = e_i$, for $1 \leq i \leq n$.

As an example of type inference using rule ($\texttt{LETREC}^{\circ}$), consider the following definitions (analogous to the ones defining *map*, *squareList* and *complementList*):

$$m = \lambda f.\, \lambda x.\, f\, x \qquad x = m\, (\lambda x.\, 0)\, 1 \qquad y = m\, (\lambda x.\, \texttt{'0'})\, \texttt{'1'}$$

We infer that (where e_m, e_x and e_y are the expressions defining m, x and y):

$$\begin{array}{l}\emptyset \vdash e_m\colon ((\alpha\to\beta)\to\alpha\to\beta,\emptyset)\\ \emptyset \vdash e_x\colon (\beta_1,\{m\colon(\alpha_1\to\mathtt{int})\to\mathtt{int}\to\beta_1\})\\ \emptyset \vdash e_y\colon (\beta_2,\{m\colon(\alpha_2\to\mathtt{char})\to\mathtt{char}\to\beta_2\})\end{array}$$

According to rule (LETREC°), we have that $\ell = 4$. Substitution S_1 is given by

$$S_1 = unify\left(\left\{\begin{array}{l}(\alpha'\to\beta')\to\alpha'\to\beta' \;\;= (\alpha_1\to\mathtt{int})\to\mathtt{int}\to\beta_1,\\ (\alpha''\to\beta'')\to\alpha''\to\beta'' = (\alpha_2\to\mathtt{char})\to\mathtt{char}\to\beta_2\end{array}\right\}\right)\circ id$$

Then $S_1\Gamma_\Diamond = \Gamma_\Diamond$ and, since also $S_1\tau_i = \tau_i$, for each τ_i $(i = 1,\ldots,3)$ inferred above, it is easy to see that further unifications are not needed. This condition is often verified and can clearly be used as an optimization. Thus, $S' = S_1$ and we have that $S'\Gamma_\Diamond = \Gamma_\Diamond$ and $\sigma_m = \forall\alpha\,\beta.(\alpha\to\beta)\to\alpha\to\beta$. Since $\sigma_m \preceq_{id} \tau'$, for each $m : \tau' \in S_1\Gamma_\Diamond$, the definitions are well-typed.

As another example consider type inference for the following definitions:

$$h = \lambda x.\,(g\,x)\;\;\mathtt{+}\;\;\mathtt{1} \qquad\qquad g = \lambda x.\,h\,(g\,x)$$

We infer that (where e_h and e_g are the expressions defining h and g, respectively):

$$\begin{array}{l}\emptyset \vdash e_h : (\alpha\to\mathtt{int},\{g:\alpha\to\mathtt{int}\})\\ \emptyset \vdash e_g : (\alpha_1\to\beta_2,\{h:\beta_1\to\beta_2, g:\alpha_1\to\beta_1\})\end{array}$$

By rule (LETREC°) we have that $\ell = 3$. Substitutions S_1 and S_2 are given by

$$S_1 = unify\left(\left\{\begin{array}{l}\alpha'\to\mathtt{int} = \beta_1\to\beta_2,\\ \alpha'_1\to\beta'_1 \;= \alpha\to\mathtt{int},\\ \alpha''_1\to\beta''_2 = \alpha_1\to\beta_1\end{array}\right\}\right)\circ id$$

$$S_2 = unify\left(\left\{\begin{array}{l}\alpha'\to\mathtt{int} \;= \beta_1\to\mathtt{int},\\ \alpha'_1\to\mathtt{int} = \alpha\to\mathtt{int},\\ \alpha''_1\to\mathtt{int} = \alpha_1\to\beta_1\end{array}\right\}\right)\circ S_1$$

and we have that $S_3 = id\circ S_2$. Moreover, we have that $S' = S_3\circ id$ and thus $S'\Gamma_\Diamond = \{h : \mathtt{int}\to\mathtt{int}, g : \alpha\to\mathtt{int}, g : \alpha_1\to\mathtt{int}\}$. Also, $\sigma_h = \forall\alpha.\,\alpha\to\mathtt{int}$ and $\sigma_g = \forall\alpha.\,\alpha\to\mathtt{int}$. Since $\sigma_h \preceq_{id} \tau'$ for all $h : \tau' \in S'\Gamma_\Diamond$ and $\sigma_g \preceq_{id} \tau'$ for all $g : \tau' \in S'\Gamma_\Diamond$, each of the mutual recusive definitions is well typed. Note that:

i) $\Gamma_{\mathrm{p}} = \{h : \mathtt{int}\to\mathtt{int}, g : \alpha\to\mathtt{int}, g : \alpha_1\to\mathtt{int}\}$ is the smallest typing context that can be used to derive a type for these mutually recursive definitions. Typing context $\{h : \beta\to\mathtt{int}, g : \alpha\to\mathtt{int}, g : \alpha_1\to\mathtt{int}\}$ (or any other typing context smaller that Γ_{p}) cannot be used to derive any type whatsover for these definitions.

ii) The greatest type for g and h, $\forall\alpha.\,\alpha\to\mathtt{int}$ can be derived with Γ_{p}.

These remarks are instances of Theorem (PRINCIPAL TYPING) below.

Theorem 7 (Soundness). *Let Γ_0 be an* ML *typing context. If $\Gamma_0 \vdash^{\circ} e{:}(\tau, \Gamma)$ is provable in* $\mathtt{T_o}$+(FIX$^{\circ}$)+(LETREC$^{\circ}$)*, then $\Gamma \vdash' e : \tau$ is provable in* ML′+(FIX′)+(LETREC′)*.*

Theorem 8 (Principal Typing). *Let (e, Γ_0) be a typing problem such that Γ_0 is an* ML *typing context. If $\Gamma_0|_{fv(e)} \vdash^{\circ} e{:}(\tau_{\mathrm{p}}, \Gamma_{\mathrm{p}})$ is provable in* $\mathtt{T_o}$+(FIX$^{\circ}$)+(LETREC$^{\circ}$)*, then $(\sigma_{\mathrm{p}}, \Gamma_{\mathrm{p}})$ is the principal typing solution for typing problem (e, Γ_0) in* ML′+(FIX′)+(LETREC′)*, where close$(\tau_{\mathrm{p}}, \sigma_{\mathrm{p}}, tv_{\mathrm{u}}(\Gamma_{\mathrm{p}}))$; otherwise, (e,Γ_0) has no solution in* ML′+(FIX′)+(LETREC′)*.*

6 Related Work

Shao and Appel [SA93] presented a smartest recompilation system for Standard ML that is also based on Damas's algorithm T. They do not address the problem of type inference for mutually recursive definitions, nor the problem of separate compilation of mutually recursive modules, since top-level declarations in the Standard ML module language cannot be mutually recursive.

Aditya and Nikihil [AN91] also used a similar type inference algorithm in an incremental compiler for the language Id. Their algorithm does not infer principal typings and they use rule (LETREC-M) for typing mutually recursive definitions. As a consequence, the addition of a new definition may cause the entire program to be recompiled, since this may require that the call graph of the program is examined to define an order in which to type the given definitions.

Jim [Jim96] adresses applications of the principal typing property and presents a type system based on rank2 intersection types that has principal typings and can be restricted to type core-ML expressions. He suggested that the given type inference algorithm could be used as a basis for the implementation of a separate compilation system for languages based on ML-like type inference. He also discusses the problem of type inference for mutually recursive definitions, but the rules presented in his work for typing recursive definitions can only type the same expressions that are typed with rule (FIX-M) used in ML.

The result of undecidability of type inference for ML+(FIX-P) was proved independently by Henglein [Hen93] and Kfoury et al.[KTU93]. It is a corollary of the result of undecidability of the semi-unification problem [KTU90], obtained by showing that the semi-unification problem is polynomial-time reducible to typability in ML+(FIX-P). Our rule (FIX′) is based essentially on the same idea of the rule proposed in [Myc84], that uses a kind of transformation from infinitary to finitary polymorphism, in a similar way to (rank 2) type systems of intersection types (which have decidable type inference of principal typings [KW99]). This avoids typability of expressions to be constrained by the solution of a set of semi-unification inequations, generating, instead, constraints that are expressed as a limited sequence of unification equations between simple types.

7 Conclusion

We have presented a type system (ML′) and a type inference algorithm ($\mathtt{T_o}$) for typing core-ML expressions extended with mutually recusive let-bindings. The

new rule for typing mutually recursive definitions can type more expressions than the corresponding rule used in ML, allowing mutually recursive definitions to be used polymorphically by other definitions. This eliminates the need to examine the call graph of a program to determine an order in which to type definitions and provides support for compilation of mutually recursive modules.

The idea behind our rule for typing recursive definitions is essentially the same as the one proposed by Mycroft. However, the extensions necessary to define rules for typing and for inferring types of mutually recursive definitions, presented in this paper, are certainly non-trivial.

A prototype implementation of type inference algorithm $\mathtt{T}_o$ is available at `http://www.dcc.ufmg.br/~camarao/MLo`.

References

[AN91] Shail Aditya and Rishyiur Nikhil. Incremental polymorphism. In *Functional Prog. Lang. and Computer Arch.*, number 523 in LNCS, pages 379–405, 1991.

[Dam84] Luís Damas. *Type Assignment in Programming Languages*. PhD thesis, Edinburgh, 1984.

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

[Hen93] Fritz Henglein. Type Inference with Polymorphic Recursion. *ACM TOPLAS*, 15(2):253–289, April 1993.

[Jim96] Trevor Jim. What are principal typings and what are they good for? In *Conference Record of POPL'96*, pages 42–53, 1996.

[KTU90] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. In *Proc. of the 22nd Annual ACM Symp. on Theory of Computation (STOC)*, pages 468–476. ACM Press, 1990.

[KTU93] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM TOPLAS*, 15(2):290–311, 1993.

[KTU94] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An Analysis of ML Typability. *Journal of the ACM*, 41(2):368–398, 1994.

[KW94] A. J. Kfoury and J. B. Wells. A Direct Algorithm for Type Inference in the Rank-2 Fragment of the Second-Order λ-Calculus. In *Proc. of the 1994 ACM Conference on LISP and Functional Prog.*, pages 196–207, 1994.

[KW99] A. J. Kfoury and J. B. Wells. Principality and Decidable Type Inference for Finite-Rank Intersection Types. In *POPL'99 ACM-SIGPLAN Symposium on the Principles of Programming Languages*. ACM Press, 1999.

[Mee93] Lambert Meertens. Incremental polymorphic type checking in B. In *10th ACM POPL*, pages 265–275. ACM, 1993.

[MH93] John Mitchell and Robert Harper. On the type structure of standard ML. *ACM TOPLAS*, 2(15):211–252, 1993.

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

[Myc84] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the International Symposium on Programming*, volume 167 of *LNCS*, pages 217–239, 1984.

[SA93] Zhong Shao and Andrew W. Appel. Smartest recompilation. In *20th ACM POPL*, pages 439–450. ACM Press, 1993.