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 ($\mathtt{ML}$) has principal types, but not principal typings. Damas also defined in his thesis a slightly modified version of $\mathtt{ML}$, that we call $\mathtt{ML}'$, which, given a typing context and an expression, derives exactly the same types, and provided an algorithm (named as $\mathtt{T}$) that infers principal typings for $\mathtt{ML}'$. This work extends each of $\mathtt{ML}'$ and $\mathtt{T}$ 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]. and compilation of partially developed modules (modules which include As he suggests, principal typings are also 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 $\mathtt{ML}'$, uses exactly the same syntax of types of the Damas-Milner system ($\mathtt{ML}$), 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 $\mathtt{ML}'$, which he named algorithm $\mathtt{T}$.

This work extends $\mathtt{ML}'$ and $\mathtt{T}$ with a new rule for typing mutually recursive let-bindings. The proposed rule can type more expressions than the corresponding rule used in ML, allowing 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 $\mathtt{ML}$ types. Section 4 presents type system $\mathtt{ML}'$ and algorithm $\mathtt{T}_o$ (a slightly modified version of algorithm $\mathtt{T}$), which infers principal typings for $\mathtt{ML}'$. Section 5 extends $\mathtt{ML}'$ and $\mathtt{T_o}$ for typing mutually recursive 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 $\Gamma = \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$, also written as $\{x_i : \sigma_i\}^{i=1..n}$, then $dom(\Gamma) = \{x_1, \dots, x_n\}$. 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 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 5 (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 6 (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]

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

An equivalent version of `ML` type system [Hen93, KTU93, KTU94, CDDK86] 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 $close(\tau, \sigma, tv(\Gamma))$.

[1] As usual (c.f. [MH93, Hen93, 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 (c.f. [CDDK86]).

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

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

$$\frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x.\, e : \tau' \to \tau} \tag{ABS}$$

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

Figure 1: Type System `ML`

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 $close(\sigma,\sigma') = close(\sigma,\sigma',\emptyset)$.

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 simple types).

Meta-variable S is used to range over substitutions; $S\sigma$ 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 7 (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 $S = S_2 \circ S_1$

[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.

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 $\beta \to (\alpha_1 \to \alpha_2)$.

According to Def. 3, we have, e.g., $\{x : \alpha, y : \beta\} \preceq_S \{x : \alpha, y : \alpha\}$, where $S(\alpha) = \alpha$ and $S(\beta) = \alpha$. This example illustrates that $\preceq_S$ is not symmetric. As another simple example, $\{x : \alpha\} \preceq_S \{x : \tau\}$, for any α, τ and S such that $S(\alpha) = \tau$. This is based on $\tau \preceq_S \alpha$ and $\emptyset \preceq_S \emptyset$. As an example of the induced ordering on typings, we have that $(\alpha, \{x : \alpha, y : \beta\}) \preceq (\alpha, \{x : \alpha, y : \alpha\})$, since $\{x : \alpha, y : \beta\}) \neq \{x : \alpha, y : \alpha\}$ and $\{x : \alpha, y : \beta\}) \preceq \{x : \alpha, y : \alpha\}$.

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: $\{x : \forall\alpha.\,\alpha\}, \{x : \forall\alpha.\,\alpha \to \alpha\}, \{x : \forall\alpha.\,(\alpha \to \alpha) \to (\alpha \to \alpha)\}, \ldots$

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 $tv_{\mathrm{u}}(\Gamma)$, the set of variables of the types of all λ-bound variables in the typing context Γ, as $tv_{\mathrm{u}}(\Gamma) = tv(\{u : \sigma \in \Gamma\})$.

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

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

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

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

Figure 2: Type System $\mathtt{T_o}$

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:

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

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.

Algorithm $\mathtt{T_o}$, defined in Fig.2, infers principal typings for ML$'$. $\mathtt{T_o}$ 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 ML$'$, $\mathtt{T_o}$ 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:

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

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

Function unify, used in rules ($\mathrm{LET}^{\mathrm{o}}$) and ($\mathrm{APPL}^{\mathrm{o}}$), computes the most general unifier of the set of types of each lambda-bound variable in the inferred typing contexts.

Notation $\mathcal{E}_{\mathrm{u}}(\Gamma,\Gamma')$ represents the set of equations on the types of each lambda-bound variable occurring both in Γ and Γ', defined as:

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

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

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

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

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

To state the principal typing property of $\mathtt{T}_{\mathrm{o}}$ with respect to $\mathtt{ML}'$, we must first define an ordering on typing contexts which takes into account the fact that typing contexts may contain more than one assumption for the same variable. This is defined in terms of the following ordering on sets of types:

Definition 8 (Ordering on sets of types). Given partial order $\preceq$ on types, the partial order $\preceq$ on sets of types is defined as follows:

$$\{\sigma_i\}^{i=1..n} \preceq \{\sigma'_j\}^{j=1..m} \quad \begin{array}{l}\text{if, for each } \sigma'_j,\ j = 1..m,\\ \quad \text{there exists } \sigma_i,\ 1 \leq i \leq n,\\ \quad \text{such that } \sigma_i \preceq \sigma'_j\end{array}$$

According to Definition 8 and to our previous definition of the ordering on typing contexts, we have, for example, that

$$\{x\!:\!\mathit{Int}, x\!:\!\mathit{Float}\} \preceq_{id} \{x\!:\!\mathit{Int}, x\!:\!\mathit{Float}, x\!:\!\mathit{Bool}, x\!:\!\alpha\} \preceq_{id} \{x\!:\!\forall\alpha.\alpha\}$$

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

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

Theorem 4 (Principal Typing $\mathtt{T}_{\mathrm{o}}$-$\mathtt{ML}'$). Let (e, Γ_0) be a typing problem such that $\Gamma_0|_{fv(e)}$ is an $\mathtt{ML}$ typing context. If $\Gamma_0 \vdash^{\mathrm{o}} e : (\tau, \Gamma)$ is provable, then (σ_p, Γ_p) is the principal typing solution for (e, Γ_0) in $\mathtt{ML}'$, where $\Gamma_p = \Gamma|_{fv(e)}$ and $\mathit{close}(\tau, \sigma_p, tv_{\mathrm{u}}(\Gamma_p))$; otherwise, (e, Γ_0) has no solution in $\mathtt{ML}$.

5 Mutual Recursion

The language of core-ML is extended with mutually recursive definitions, by including expressions of the form: $\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e$, which we also write as $\texttt{letrec}\ \{x_i = e_i\}^{i \in I}\ \texttt{in}\ e$, where all x_i are distinct.

ML uses the following rule to type mutual recursion [Myc84, Jim96]:

$$\frac{\begin{array}{c}\forall j \in I\ \ \Gamma, \{x_i : \tau_i\}^{i \in I} \vdash e_j : \tau_j \\ \Gamma, \{x_i : \sigma_i\}^{i \in I} \vdash e : \tau\end{array}}{\Gamma \vdash \texttt{letrec}\ \{x_i = e_i\}^{i \in I}\ \texttt{in}\ e : \tau} \quad \forall i \in I,\ close(\tau_i, \sigma_i, tv(\Gamma)) \qquad \text{(LETREC-M)}$$

In this rule, recursive definitions must be typed under the assumption that the recursive variables have simple types. In typing the body of `letrec`, the types of the recursive variables are generalized, enabling the variables to be used polymorphicaly. A well-known limitation of this rule is illustrated by the following example, given in [Myc84]:

$$\begin{array}{lll} map & = & \lambda f.\,\lambda l.\ \texttt{if}\ null\,l\ \texttt{then}\ nil\ \texttt{else}\ f\ (hd\,l)\ \texttt{::}\ map\ f\ (tl\,l) \\ squarelist & = & \lambda l.\ map\ (\lambda x.\,x \times x)\ l \\ complement & = & \lambda l.\ map\ (\lambda x.\,not\,x)\ l \end{array}$$

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 $\texttt{letrec}\ B_1\ \texttt{in}\ \ldots(\texttt{letrec}\ B_m\ \texttt{in}\ 0)$ 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.

We give next a new rule (LETREC') for typing mutually recursive definitions, that allows recursive variables to be used polymorphically by other definitions, eliminating the need to examine the call graph of a program to determine an order in which to type definitions.

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

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

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

Figure 3: Typing rules for polymorphic recursion

Consider, for example, a type derivation for the previously given definitions of *map* (e_m), *squarelist* (e_s) and *complement* (e_c). Letting

$$\begin{array}{l} \Gamma_0 = \{(::) : \forall\alpha.\alpha \to [\alpha] \to [\alpha],\ hd : \forall\alpha.[\alpha] \to \alpha,\ tl : \forall\alpha.[\alpha] \to [\alpha]\} \\ \Gamma' = \{map : (\mathtt{int} \to \mathtt{int}) \to [\mathtt{int}] \to [\mathtt{int}], \\ \qquad map : (\mathtt{bool} \to \mathtt{bool}) \to [\mathtt{bool}] \to [\mathtt{bool}]\} \end{array}$$

we derive

$$\begin{array}{l} \Gamma_0, \Gamma' \vdash e_m : (\alpha \to \beta) \to [\alpha] \to [\beta] \\ \Gamma_0, \Gamma' \vdash e_s : [\mathtt{int}] \to [\mathtt{int}] \\ \Gamma_0, \Gamma' \vdash e_c : [\mathtt{bool}] \to [\mathtt{bool}] \end{array}$$

and have that $\sigma_{map} = \forall\alpha.\forall\beta.(\alpha \to \beta) \to [\alpha] \to [\beta]$, $\sigma_{map} \preceq_{id} (\mathtt{int} \to \mathtt{int}) \to [\mathtt{int}] \to [\mathtt{int}]$ and $\sigma_{map} \preceq_{id} (\mathtt{bool} \to \mathtt{bool}) \to [\mathtt{bool}] \to [\mathtt{bool}]$.

`ML` uses rule (FIX-M), given in Fig.3, for typing recursive expressions of the form $\mu x.\, e$, representing a program x such that $x = e$, where e may contain occurrences of x. Due to the limitations of this rule, which requires that all recursive calls in the body of a recursive function have the same *monomorphic* type as the enclosing function, Mycroft [Myc84] and Meertens [Mee93] proposed, independently, rule (FIX-P), also presented in Fig.3.

As an example, expression $\mu x.\, xx$, which is not typable in `ML+`(FIX-M), is typeable in `ML+`(FIX-P). For a more practical example, consider:

```
datatype α Seq = NIL' | CONS' of α × (α × α) Seq
fun sizeS NIL' = 0
  | sizeS (CONS'(x,ps)) = 1 + 2 * sizeS ps
```

Function *sizeS* would be rejected by Standard `ML`, which uses rule (FIX-M), since it violates the restriction imposed in this rule, because the outer *sizeS* has type $\alpha\ Seq \to \mathtt{int}$ whereas the inner *sizeS* has type $(\alpha \times \alpha)\ Seq \to \mathtt{int}$.

An analogous rule to rule (FIX-P) for typing simple recursive definitions can be defined to type mutually recursive definitions:

$$\frac{\begin{array}{c} \forall j \in I\ \ \Gamma, \{x_i : \sigma_i\}^{i\in I} \vdash e_j : \tau_j \\ \Gamma, \{x_i : \sigma_i\}^{i\in I} \vdash e : \tau \end{array}}{\Gamma \vdash \mathtt{letrec}\ \{x_i = e_i\}^{i\in I}\ \mathtt{in}\ e : \tau} \quad \forall i \in I,\ close(\tau_i, \sigma_i, tv(\Gamma)) \quad \text{(LETREC-P)}$$

Noting that `let` $x = \mu x.e$ `in` x is equivalent to `letrec` $x = e$ `in` x, we can directly derive, from rule (LETREC$'$), a simpler rule (FIX$'$) for typing recursive expressions of the form $\mu x.e$, which is also presented in Fig.3.

The relation between typability in ML$'$+(FIX$'$)+(LETREC$'$) and typability in ML+(FIX-P)+(LETREC-P) is formally stated by Theorems 5 and 6 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).

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):

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

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

Type inference for the extension of ML with rule (FIX-P), often called *Milner-Mycroft Calculus*, was shown to be undecidable, by Heiglein [Hen93] and by Kfoury et al. [KTU93]. On the other side, type inference for ML$'$+(FIX$'$) is decidable, as we show below. Before trying to elucidate this apparent contradiction, we present first a type inference rule corresponding to rule (FIX$'$), and give some simple examples, deferring the discussion of this contradiction to Section.6.

Type inference algorithm $\mathtt{T_o}$ is extended with rule (FIX$^{\mathrm{o}}$), given in Fig.4, for inferring types of expressions of the form $\mu x.\, e$. Notation $\mathcal{E}_{\mathrm{p}}\,(\tau, \{\tau_i\}^{i=1..n}, V)$, used in this rule, represents the set of type equations defined as follows, where $\{\alpha_j\}^{j=1..m} = tv(\tau) - V$ and $\{\alpha_{ij}\}^{i=1..n, j=1..m,}$ are fresh type variables:

$$\mathcal{E}_{\mathrm{p}}\,(\tau, \{\tau_i\}^{i=1..n}, V) = \{\tau_i' = \tau_i \mid \tau_i' = \tau[\alpha_{ij}/\alpha_j]^{j=1..m}\}^{i=1..n}$$

Type inference in ML$'$+(FIX$^{\mathrm{o}}$) is illustrated by some simple examples below.

Consider type inference for expression $\mu x.\, x x$, given an empty typing context. We have that $\emptyset \vdash x\ x : (\beta, \{x : \alpha \to \beta, x : \alpha\})$. According to rule (FIX$^{\mathrm{o}}$), substitution S is obtained as $id \circ S_0$, where

$$S_0 = unify(\{\beta' = \alpha \to \beta,\ \beta'' = \alpha\})$$

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

$$\frac{\Gamma_0 \ominus x \vdash e : (\tau, \Gamma)}{\Gamma_0 \vdash \mu x.\, e : (S\tau, S(\Gamma \ominus x))} \quad \begin{array}{l} S_0 = \mathit{unify}(\mathcal{E}_{\mathrm{p}}(\tau, \Gamma(x), tv_{\mathrm{u}}(\Gamma))) \\ S = \mathit{unify}(\mathcal{E}_{\mathrm{u}}(\Gamma, S_0\Gamma)) \circ S_0 \\ \mathit{close}(S\tau, \sigma, tv_{\mathrm{u}}(S\Gamma)) \\ \sigma \preceq_{id} \tau', \text{ for each } \tau' \in S\Gamma(x) \end{array} \quad (\mathrm{FIX^o})$$

Figure 4: Type inference rule for recursive expressions

As another illustrative example, consider type inference for the following definition, in an empty typing context:

$$h = \lambda x.\ \texttt{if}\ h\,x \texttt{==} 1\ \texttt{then}\ x\ \texttt{else}\ h\,x$$

We have that $\emptyset \vdash e_h : (\alpha \to \alpha, \{h : \alpha \to \texttt{int}, h : \alpha \to \alpha\})$. By rule ($\mathrm{FIX^o}$) we have that $S = id \circ S_0$, where

$$S_0 = \mathit{unify}(\{\alpha' \to \alpha' = \alpha \to \texttt{int},\ \alpha'' \to \alpha'' = \alpha \to \alpha\})$$

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

As a simple example of a recursive definition that is not typeable, consider $f = \lambda x.f$. We have that $\emptyset \vdash \lambda x.f : (\alpha \to \beta, \{f : \beta\})$. Then $S = id \circ S_0$, where

$$S_0 = \mathit{unify}(\{\alpha' \to \beta' = \beta\}) = id \dagger \{\beta \mapsto (\alpha' \to \beta')\}$$

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

The reader can verify that the proposed rules also apply to the type inference for nested mutually recursive expressions, such as

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

Algorithm $\mathtt{T_o}$ is extended with rule ($\mathrm{LETREC^o}$), given in Fig.5, for inferring types of mutually recursive definitions.

As a first example of application of this rule, consider type inference for the following definitions (analogous to the example defining *map*, *squarelist* and *complement* at the beginning of this section):

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

We have that

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

Substitution S_1 is given by

$$S_1 = \mathit{unify}\left(\left\{ \begin{array}{ll} (\alpha' \to \beta') \to \alpha' \to \beta' & = (\alpha_1 \to \texttt{int}) \to \texttt{int} \to \beta_1, \\ (\alpha'' \to \beta'') \to \alpha'' \to \beta'' & = (\alpha_2 \to \texttt{char}) \to \texttt{char} \to \beta_2 \end{array} \right\}\right)$$

$$\frac{\begin{array}{c}\forall i \in I \;\; \Gamma_0 \ominus \{x_i\}^{i\in I} \vdash e_i : (\tau_i, \Gamma_i) \\ S'\Gamma_\Diamond, \{x_i : \sigma_i\}^{i\in I} \vdash e : (\tau, \Gamma)\end{array}}{\Gamma_0 \vdash \texttt{letrec } \{x_i = e_i\}^{i\in I} \texttt{ in } e : (S\tau, \Gamma')} \quad (\text{LETREC}^{\text{o}})$$

where $\Gamma_\Diamond = \bigcup_{i\in I} \Gamma_i$

$$\begin{array}{l}
S_0 = id \\
\forall j \in I,\;\; S_j = \mathit{unify}\left(\bigcup_{i\in I} \mathcal{E}_{\mathrm{p}}(S_{j-1}\tau_i, S_{j-1}\Gamma_\Diamond(x_i), tv_{\mathrm{u}}(S_{j-1}\Gamma_i))\right) \circ S_{j-1} \\
S' = \mathit{unify}\left(\bigcup_{i,j\in I, i\neq j} \mathcal{E}_{\mathrm{u}}(S_n\Gamma_i, S_n\Gamma_j) \cup \mathcal{E}_{\mathrm{u}}(S_n\Gamma_\Diamond, \Gamma_0)\right) \circ S_n \\
\forall i \in I,\;\; close(S'\tau_i, \sigma_i, tv_{\mathrm{u}}(S'\Gamma_i)), \\
\forall i \in I,\;\; \sigma_i \preceq_{id} \tau',\; \text{for each } \tau' \in S'\Gamma_\Diamond(x_i) \\
S = unify(\mathcal{E}_{\mathrm{u}}(S'\Gamma_\Diamond, \Gamma)) \\
\Gamma' = (S\Gamma \cup SS'\Gamma_\Diamond) \ominus \{x_i\}^{i\in I}
\end{array}$$

Figure 5: Type inference rule for mutually recursive definitions

Then $S_1\Gamma_\Diamond = \Gamma_\Diamond$ and, since also $S_1\tau_i = \tau_i$ for each τ_i 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. Also, $\sigma_m = \forall\alpha\,\beta.(\alpha \to \beta) \to \alpha \to \beta$ and, since $\sigma_m \preceq_{id} \tau'$, for each $m : \tau' \in S_1\Gamma_\Diamond$, the definitions are well-typed.

The following example illustrates the need of at most n unifications in rule (LETREC$^{\text{o}}$) (where n is the number of definitions). Consider type inference for the following set of mutually recursive definitions:

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

We have that

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

Substitution S_1 is obtained as

$$S_1 = unify\left(\left\{\begin{array}{l}\alpha' \to \texttt{int} = \beta_1 \to \beta_2, \\ \alpha_1' \to \beta_2' \;= \alpha \to \texttt{int}, \\ \alpha_1'' \to \beta_2'' = \alpha_1 \to \beta_1\end{array}\right\}\right)$$

We have that $S_1\Gamma_\Diamond = \{h : \beta_1 \to \texttt{int}, g : \alpha \to \texttt{int}, g : \alpha_1 \to \beta_1\}$, and also that $S_1\tau_h = \alpha \to \texttt{int}$ and $S_1\tau_g = \alpha_1 \to \texttt{int}$. Substitution S_2 is thus obtained as

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

Then $S_2\Gamma_\Diamond = \{h : \texttt{int} \to \texttt{int}, g : \alpha \to \texttt{int}, g : \alpha_1 \to \texttt{int}\}$, $\sigma_h = \forall\alpha.\,\alpha \to \texttt{int}$ and $\sigma_g = \forall\alpha.\,\alpha \to \texttt{int}$. Since $\sigma_h \preceq_{id} \tau'$ for all $h : \tau' \in S_2\Gamma_\Diamond$ and $\sigma_g \preceq_{id} \tau'$ for all $g : \tau' \in S_2\Gamma_\Diamond$, each of the mutual recusive definitions is well typed.

Soundness and principal typing theorems completely analogous to theorems 3 and 4 can also be proved to be valid for $\mathtt{ML}'$ extended with rules (FIX$'$) and (LETREC$'$) and $\mathtt{T_o}$ extended with rules (FIX^o) and (LETREC^o).

6 Related Work

Shao and Appel [SA93] presented a smartest recompilation system for Standard ML that is also based on Damas's algorithm $\mathtt{T}$. They do not address the problem of type inference for mutually recursive definitions, nor the problem of separate compilation of mutually recursive modules.

Aditya and Nikihil [AR91] 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.

Jim [Jim96] presented a type system based on rank2 intersection types that has principal typings and can be restricted to type core-ML expressions. He also discusses type inference for mutually recursive definitions, but the rules presented in his work for typing mutually recursive definitions can only type the same expressions that are typed with rule (FIX-M) used in $\mathtt{ML}$.

Type inference in $\mathtt{ML}$+(FIX-P) has been proved to be undecidable [Hen93, KTU93]. As mentioned previously, our rule for typing polymorphic recursion is based on the same idea as the rule proposed in [Myc84]. This 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]). Our conjecture is that this is the main reason that enabled us to achieve decidable type inference for mutually recursive definitions.

7 Conclusion

We have presented a type system ($\mathtt{ML}'$) 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, by 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, providing support for compiling mutually recursive modules in languages that use ML-like type inference.

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

[AR91] Shail Aditya and Rishyiur. Incremental polymorphism. In *Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 379–405. Springer-Verlag, 1991.

[CDDK86] D. Clement, D. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative Language Mini-ML. In *ACM Symp. on Lisp and Functional Programming*, pages 13–27, 1986.

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

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

[Hen93] Fritz Henglein. Type Inference with Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[Jim96] Trevor Jim. What are principal typings and what are they good for? In *Conf. Record of POPL'96: the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–53, 1996.

[KTU93] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 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 Programming*, 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*, 1999.

[Mee93] Lambert Meertens. Incremental polymorphic type checking in B. In *10th ACM Symposium on Principles of Programming Languages (POPL)*, pages 265–275. ACM, 1993.

[MH93] John Mitchell and Robert Harper. On the type structure of standard ML. *ACM Transactions on Programming Languages and Systems*, 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. Springer-Verlag, 1984.

[SA93] Zhong Shao and Andrew W. Appel. Smartest recompilation. In *20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 439–450, 1993.