

ML Has Principal Typings

Carlos Camarão and Lucília Figueiredo

Abstract

Is there a type system for core-ML that, using exactly the same syntax of types of the Damas-Milner system, types exactly the same terms of this system, and has principal typings?

In this article we answer this question affirmatively.

A definition of principal typing is given, capturing the simple idea of representing the set of all typings that can be obtained in derivations for a given term in a given type system. This definition is parameterised on an ordering on types, enabling it to be used for different type systems.

A type system for core-ML is presented that uses type expressions with the same form as the Damas-Milner system, and considers as well-typed the same expressions of the Damas-Milner system. A type inference algorithm is then presented, which computes principal typings with respect to the given type system.

1 Introduction

This article answers affirmatively the following question. Consider the language of core-ML[Mil78,MH88,MH93], where terms have the simple context-free form given by $e ::= x \mid \lambda x. e \mid e e' \mid \text{let } x = e \text{ in } e'$, with meta-variable x ranging over a countably infinite set of *variables*, and consider that types of these terms have the simple syntax given by $\tau ::= \alpha \mid \tau \rightarrow \tau'$, and $\sigma ::= \forall \alpha. \sigma \mid \tau$, where meta-variable α ranges over a countably infinite set of *type variables*, disjoint from the set of (term) variables; then, is there a type system that types exactly the same terms as the Damas-Milner system[Mil78,DM82], and has principal typings?

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 and because their presence is not relevant in this work (c.f. [CDDK86]).

As well known, core-ML is a simple extension of the simply-typed lambda calculus[Chu40,CF58,Hin69] with let-bindings, where variables introduced in these bindings may be used in contexts requiring expressions of different types. Such variables are said to have quantified (or polymorphic) types; the contrast with variables introduced in lambda-abstractions, which may not be used in

contexts requiring expressions of different types, is a distinctive characteristic of core-ML. Thus, an important simplification of the language of types of the Damas-Milner system, which adequately describes the context-sensitive syntax of core-ML, is that the universal quantifier can only appear as prefixes of types (a critical difference with respect to the impredicative second-order lambda-calculus, also called system F[Gir71,Gir72,Rey74], and other predicative polymorphic systems based on the second-order lambda calculus).

This work is concerned with the concept of *principal typing*[Dam84,Jim96], not *principal type*. As pointed out by Damas[Dam84] and emphasized by Jim[Jim96], the Damas-Milner system has principal types, but not principal typings. In this paper we show that this is a property of the Damas-Milner system, not of core-ML, in contrary to what is currently believed[Jim96]. We present a type system ML' and an algorithm ML_o that infers principal typings for ML' . Type system ML' considers as well-typed exactly the same terms as the Damas-Milner system, and uses type expressions with exactly the same form as in the Damas-Milner system.

Some readers might draw an analogy to the situation that occurs when a type system for the simply-typed lambda-calculus is extended with subtypes, a subsumption rule, and subtyping constraints[FM90,Mit88]. Fuh and Mishra[FM90], as well as Mitchell[Mit88], use subtyping constraints in typing contexts to recover the principal type property for the simply-typed lambda-calculus extended with subtypes and a subsumption rule.¹ However, core-ML has principal types, as shown by Damas and Milner[Mil78,DM82]. We show in this paper that it does have also principal typings, without the need for extending the language of types with intersection types or some form of constrained types.² We require that the context-free form of terms be that of core-ML and the simple language of types be the one used in the Damas-Milner system.

The definition of whether a typing for a given expression is principal or not should be based on a partial order on typings, which in turn should be based on both an ordering of types and on an ordering of typing contexts. The definitions given in this paper do exactly that. Somewhat unfortunately, our

¹ As shown by Fuh and Mishra, in the presence of the simplest possible subtype relation, defining that one type constant is a subtype of another — say, `int` is a subtype of `real` — the principal type for $\lambda x. x$ in the empty context, which should be of the form $\alpha \rightarrow \alpha$, cannot be used to obtain, by substitution of types for type variables, type `int` \rightarrow `real`, which can be derived for $\lambda x. x$. Fuh and Mishra also show that it is not enough to redefine the principal type property, by considering principal types as representing instances, obtained by substitution, or supertypes of these instances. This can be seen by considering for example function `twice`, where `twice` $\equiv \lambda f. \lambda x. f(f x)$. Due to the contravariance of the function type constructor, the principal type of `twice` in the empty context, which should be of the form $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$, cannot, using this redefinition of principal type, represent type `(real` \rightarrow `int)` \rightarrow `(real` \rightarrow `int)`, which can be derived for `twice`.

² It has been shown by Jim[Jim96] that the system of rank 2 intersection types can be used to give principal typings for core-ML terms.

definition of principal typing turns out to be original. But its intent is the same as in other approaches, namely, to capture the simple idea of representing the set of all typings that can be obtained in derivations for a given expression in a given type system. The results in this paper are based on two simple points: first, on the definition of a partial orders on types, and from that on typing contexts, then on typings; and second, on the use of multiple assumptions for a variable in typing contexts. Whereas the use of multiple assumptions in typing contexts is equivalent to using intersection types in typing contexts, this work shows that intersection types are not needed for the types of expressions, in order to ‘recover’ the principal typing property for core-ML.

We do not discuss in this paper why principal typing is important and why it has different practical applications with respect to the property of principal type, for which we refer the reader to Jim’s article[Jim96]. We point out though that the principal typing property would make it possible, during program development, to specify which modules should be considered for type checking (or which modules should not), in a system with separate compilation. For names imported from modules that are not selected, the type inference algorithm would automatically infer a type general enough not to issue any type error at all on the use of these names.

In the remaining of this section we introduce some common notation and basic terminology. The following syntactic meta-variables range over the following sets of syntactic terms: x, y for variables, e for terms (expressions), σ for types, τ for simple types, α, β for type variables, and Γ for typing contexts.

To define the syntax of a language, including context-sensitive constraints, it is often necessary to use types for checking whether a given phrase is syntactically valid. For example, if (from the definition of f we can conclude that) f behaves as a function from τ to τ' (i.e. has type $\tau \rightarrow \tau'$), and (from the definition of x we can conclude that) x has type τ , then the application of f to x is well-formed, and has type τ' . Typing contexts are used to gather (type) information that can be used for enforcing context sensitive constraints.

A typing context is a finite set of pairs $x : \sigma$ (called assumptions). An assumption $x : \sigma$ is an assumption for x . If $\Gamma = \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$, also written as $\{x_i : \sigma_i\}^{i=1..n}$, then $dom(\Gamma) = \{x_1, \dots, x_n\}$, also written as $\{x_i\}^{i=1..n}$ (and similarly for other sets). We also use $\forall(\alpha_i)^{i=1..n}. \tau$ as an abbreviation for $\forall\alpha_1. \forall\alpha_2. \dots \forall\alpha_n. \tau$, where $n \geq 0$. We may drop the superscripts ($i = 1..n$), and write, for example, $\forall\alpha_i. \tau$ as an abbreviation for $\forall(\alpha_i)^{i=1..n}. \tau$.

If Γ is a typing context in which each variable 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 $\Gamma - \{x : \Gamma(x)\}$, if $x \in dom(\Gamma)$, and Γ otherwise; and $(\Gamma, x : \sigma)$ is defined as $(\Gamma \ominus x) \cup \{x : \sigma\}$.

If Γ may have more than one assumption for the same variable, then, letting $\{x : \sigma_i\}^{i=1..n}$ be the possibly empty set of all assumptions for x in Γ , we define $\Gamma(x) = \{\sigma_i\}^{i=1..n}$, $\Gamma \ominus x = ((\Gamma - \{x : \sigma_1\}) - \dots - \{x : \sigma_n\})$, and $(\Gamma, x : \sigma)$ is defined as $(\Gamma \ominus x) \cup \{x : \sigma\}$. The cardinality of $\Gamma(x)$ is denoted

by $\#\Gamma(x)$.

We call σ a quantified type if a quantifier occurs in σ , otherwise a simple type. A substitution S is a function from type variables to simple types. The identity substitution is denoted by id . The symbol \circ denotes function composition. The set of free variables of term e , denoted by $fv(e)$, and the set of free type variables of type σ , denoted by $tv(\sigma)$, have the usual definitions.

$S\sigma$ 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 $x : S\sigma$, and similarly for SX , where X is a set of types, or type variables. The operation of applying a substitution S to σ is capture-free if $tv(S\sigma) = tv(S(tv(\sigma)))$. We define $S \dagger \{\alpha \mapsto \tau\}(\beta) = S(\beta)$, if $\beta \neq \alpha$, and τ if $\beta = \alpha$, and $\sigma[\tau/\alpha] = (id \dagger \{\alpha \mapsto \tau\})\sigma$.

A type σ is closed if $tv(\sigma) = \emptyset$. Predicate $close$ is defined by $close(\tau, \sigma, V) = (\sigma = \forall \alpha_i. \tau)$, where $\{\alpha_i\} = tv(\tau) - V$. We overload $close$ to define $close(\tau, \sigma) = close(\tau, \sigma, \emptyset)$, and define also $close'(\sigma, \sigma', V) = (\sigma' = \forall \alpha_i. \sigma \text{ and } \alpha_i \subseteq V)$.

2 Typing problems and solutions, and orderings

This section defines a typing problem, a typing solution, and orderings on types, typing contexts and typings, induced by parametric polymorphism.

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. $\{\text{True} : \text{Bool}, \text{False} : \text{Bool}, 1 : \text{Int}, \dots\}$.

Definition 2 (Typing Solution) A solution to a typing problem (e, Γ_0) in a given type system is a pair (σ, Γ) 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 $(\text{not True}, \{\text{True} : \text{Bool}, \text{not} : \text{Bool} \rightarrow \text{Bool}\})$, a typing solution is $(\text{Bool}, \{\text{True} : \text{Bool}, \text{not} : \text{Bool} \rightarrow \text{Bool}\})$ (in this case this would typically be the only solution to this typing problem).

If a typing problem is an expression, and typing contexts can have quantified types, then there would exist no type error at all due to the use of a free variable (such variable might be inferred to have a type general enough so that no type error may ever be detected due to its use). This remark also applies if typing contexts can have intersection types, instead of quantified types.

A minimal element of a set with respect to a partial order R is called an R -minimal element of this set. Similarly for the smallest element of a set. We shall now proceed to give a definition of minimal (or principal) typings as the R -minimal element of a set of typings, where R is a partial order.

The ordering on types for languages with quantified and simple types formed by means of type variables and type constructors should consider relations on types obtained by substitutions of simple types for free type variables),

and relations between quantified types.

We define: (i) for each substitution S , $subs_S(\sigma, \sigma') = (S\sigma = \sigma')$; and (ii) $subb(\sigma, \sigma') = \left(\sigma = \forall \alpha. \sigma_1 \text{ and } close'((id \uparrow \alpha \mapsto \tau)\sigma_1, \sigma', tv(\tau)) \right)$.

Instantiation is a particular case of $subb$ (where $close'$ does not introduce any quantification). We use $inst(\sigma, \tau)$, meaning $subb(\sigma, \tau)$, for some simple type τ .

We can now give the definition of our partial order on types.

Definition 3 (Parametric Polymorphism) An ordering on types for languages with quantified and simple types formed by means of type variables and type constructors, which we call an *ordering induced by parametric polymorphism*, denoted by \preceq_S , is defined inductively as follows: $\sigma \preceq_S \sigma' = \text{true}$, if $subs_S(\sigma', \sigma)$, or $subb(\sigma, \sigma')$ and $S = id$, or $\sigma \preceq_{S_1} \sigma''$ and $\sigma'' \preceq_{S_2} \sigma'$ and $S = S_1 \circ S_2$; otherwise false.

We define also: $\sigma \preceq \sigma' = (\sigma \preceq_S \sigma', \text{ for some } S)$. These simple definitions will be sufficient for our purposes. More complex definitions of $subb$ and $inst$ may be necessary to allow instantiations that are not restricted to occur at the outermost level.

We have, for example: $\forall \alpha. \alpha \preceq_{id} \forall \alpha_i. \tau$, from $subb$, $\forall \alpha_i. \tau \preceq_{id} \tau$ from $subb$, and $\tau \preceq_S \alpha$ from $subs_S$, for any τ such that $tv(\tau) = \{\alpha_i\}$, and any S such that α maps to τ .

We make at this point a small deviation in the course of our main objective, in order to briefly discuss two simple properties of the ordering on types, coming directly from the given definitions.

A first property is the antimonotonicity of quantification over a subset of \preceq : if $subs_S(\sigma_1, \sigma_2)$ then $\sigma'_1 \preceq_{id} \sigma'_2$, where $close(\sigma_1, \sigma'_1)$ and $close(\sigma_2, \sigma'_2)$. Recall that $subs_S(\sigma_1, \sigma_2)$ implies $\sigma_2 \preceq_S \sigma_1$ (though in general the inverse does not hold). This property reflects, quite simply, that if “less is required” of a given expression e , then the type of the term obtained by closing e , i.e. by introducing λ -abstractions on 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 $\lambda f. f x$).

A second property is that the monotonicity of \rightarrow over $subs_S$: If $subs_S(\sigma_1, \sigma'_1)$ and $subs_S(\sigma_2, \sigma'_2)$, then $subs_S(\sigma_1 \rightarrow \sigma_2, \sigma'_1 \rightarrow \sigma'_2)$. Note that the substitutions in this rule must be the same. It follows that the function type constructor is neither monotonic nor antimonotonic, with respect to \preceq (either in the first or the second argument) .

We can conclude that parametric polymorphism is not a form of subtyping, nor vice-versa, since, in the case of subtyping, the function type constructor is antimonotonic in the first argument and monotonic in the second argument, as is well-known (see e.g. [AC96,Mit96]). This agrees with and provides a formalisation for Cardelli and Wegner’s classification of type systems[CW85]

(inasmuch as parametric polymorphism and subtyping are concerned), where these concepts are indeed parallel to each other[CW85, Figure 2]. Cardelli and Wegner consider other forms of polymorphism, apart from parametric polymorphism (for example, subtyping is a form of polymorphism, called inclusion polymorphism). We will leave for further work an analysis of the relation between subtyping and parametric polymorphism, based on their semantic models (see e.g. Mitchell[MP88,Mit96]). We point out also that principal and minimal typing both refer to the existence of a minimal³ element with respect to a partial order on typings (derivable for a given term). The same applies to principal and minimal types: both refer to the existence of a minimal element with respect to a partial order on types (derivable for a given term in a given context). Their current use differs only from which (partial) order on types is considered: a partial order induced by polymorphism (for principal typing) or subtyping (for minimal typing).⁴

A partial order on typing contexts, representing “requirements on variables” occurring in these contexts, may now be defined.

Definition 4 (Ordering on typing contexts) Given any partial order \preceq_S on types, a partial order \preceq_S on typing contexts is defined inductively as follows: $\Gamma \preceq_S \Gamma'$ means $x : \sigma \in \Gamma$ implies that there exists $x : \sigma'$ in Γ' such that $\sigma' \preceq_S \sigma$ and $\Gamma - \{x : \sigma\} \preceq_S \Gamma'$.

From this definition, $\emptyset \preceq_S \Gamma$ is vacuously true, for any Γ, S .

If $\Gamma \preceq_S \Gamma'$, then Γ *requires less of its variables than* Γ' . Informally, according to this definition Γ requires less than Γ' if for each type in Γ there is another in Γ' that provides more.

For example, $\{x : \alpha, y : \beta\} \preceq_S \{x : \alpha, y : \alpha\}$, 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 any S such that $S(\alpha) = \tau$. This is based on $\tau \preceq_S \alpha$ and $\emptyset \preceq_S \emptyset$.

We define also $\Gamma \preceq \Gamma' = \text{true}$, if $\Gamma \preceq_S \Gamma'$, for some S , and false otherwise. We can now define an ordering on typings (or typing solutions) for a given typing problem, as follows:

Definition 5 (Ordering on typings) Given a typing problem (e, Γ_0) , for any typing solutions (σ, Γ) and (σ', Γ') to this typing problem, we have: $(\sigma, \Gamma) \preceq (\sigma', \Gamma') = ((\Gamma \preceq \Gamma') \text{ and } (\Gamma = \Gamma' \text{ implies } \sigma \preceq \sigma'))$.

For example, $(\alpha, \{x : \alpha, y : \beta\}) \preceq (\alpha, \{x : \alpha, y : \alpha\})$, since both $\{x : \alpha, y : \beta\} \neq \{x : \alpha, y : \alpha\}$ and $\{x : \alpha, y : \beta\} \preceq \{x : \alpha, y : \alpha\}$.

³ In fact, ‘smallest’ would be more appropriate, to follow the common terminology used in the study of ordering relations.

⁴ Also, as pointed out for example by Mitchell[Mit96], subtyping is not a partial order in every programming language that explores this concept, since antisymmetry may not always be satisfied. However, a partial order has to be established for the existence, in general, of minimal elements.

3 Principal Typing

A definition of principal typing can now be given simply as:

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, (e, Γ) has no principal typing.

Definition 7 (Principal Type) Given a typing problem (e, Γ_0) , if there exist σ and Γ such that (σ, Γ) is a minimal typing solution to this typing problem and $\Gamma \subseteq \Gamma_0$, then σ is a *principal type* for expression e in context Γ_0 . Otherwise, there is no principal type for e in Γ_0 .

Usual definitions of principal type are equivalent to Definition 7, but are written similarly to the following: a type σ is the principal type of a given expression e in a given typing context Γ if $\Gamma \vdash e : \sigma$ is provable, and $\Gamma \vdash e : \sigma'$ is provable implies that $\sigma \preceq \sigma'$.

Jim [Jim96] stressed the difference between principal types and principal typings and pointed out that the issue has not been treated properly by a number of authors, which have published offhand claims that ML possesses the principal typing property. As mentioned by Jim, Damas and Milner proved that ML has principal types, not principal typings [Mil78,DM82]. As we show in the next two sections, however, ML does have principal typings. We define a slightly modified type system that considers as well-typed exactly the same terms as the Damas-Milner system, and for which there is a type inference algorithm that computes principal typings.

In Jim's work [Jim96], a typing problem is defined to be an expression, and the ordering on typing solutions used is as follows:

Definition 8 $(\sigma, \Gamma) \leq (\sigma', \Gamma')$ if there exists a substitution S such that $S\sigma \leq \sigma'$ and $\Gamma'(x) \leq S\Gamma(x)$, for all $x \in \text{dom}(\Gamma)$.

According to our definitions, the Damas-Milner and other similar type systems for ML do not have principal typings, as can be seen by considering that each of the following infinite list of typing contexts can be used to derive a valid type for xx in the Damas-Milner system, and for each such typing context the next one in the list is smaller: $\{x : \forall\alpha. \alpha\}$, $\{x : \forall\alpha. \alpha \rightarrow \alpha\}$, $\{x : \forall\alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)\}, \dots$

4 ML has principal typings

An equivalent version of the Damas-Milner system [DM82,KTU94,CDDK86] is presented in Figure 1. The type inference algorithm ML_p , presented in Figure 2, computes principal typings for the Damas-Milner system, for typing problems (e, Γ_0) such that Γ_0 includes all typings for variables that occur free in e and have a quantified type. For example, in ML, if x has a quantified type then expression xx is well-typed, otherwise it is not. Thus, for a typing

problem $(x x, \Gamma_0)$, typing context Γ_0 must include an assumption for x . Type system ML_p is then extended, so as to compute principal typings for typing problems of the form (e, \emptyset) , and in fact for any core-ML typing problem.

Type system ML_p is in fact a type inference algorithm. It is based on an algorithm presented by Mitchell [Mit96, Chapter 11], but avoids the use of what Mitchell calls *typing environments*, which are sets of assumptions of the form $x : (\sigma, \Gamma)$.

$\Gamma, x : \sigma \vdash x : \tau$ where $inst(\sigma, \tau)$	(VAR)
$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \sigma \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau_2}$	$close(\tau_1, \sigma, tv(\Gamma))$ (LET)
$\frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x. e : \tau' \rightarrow \tau}$	(ABS)
$\frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau}$	(APPL)

Figure 1: Type System ML

Function *unify* gives the (usual) most general unifying substitution for a set of pairs of type expressions (usually written as a set of type equations). The definition is included in the next section. We define also:

$$\begin{aligned} \Gamma|_V &= \{x : \sigma \mid x : \sigma \in \Gamma \text{ and } x \in V\} \\ s(\Gamma) &= \{x : \sigma \in \Gamma \mid \sigma \text{ is a simple type}\} \\ q(\Gamma) &= \{x : \sigma \in \Gamma \mid \sigma \text{ is a quantified type}\} \\ \mathcal{E}(\Gamma, \Gamma') &= \{\Gamma(x) = \Gamma'(x) \mid x \in dom(s(\Gamma)) \text{ and } x \in dom(s(\Gamma'))\} \\ U(\Gamma, \Gamma') &= unify(\mathcal{E}(\Gamma, \Gamma')) \end{aligned}$$

We use $\Gamma \vdash e : \tau$ and $\Gamma \vdash_p e : (\tau, \Gamma')$ for derivations in types systems ML and ML_p , respectively. The proofs of theorems are omitted, for lack of space.

Theorem 1 If $\Gamma_0 \vdash_p e : (\tau, \Gamma)$ is provable, then $\Gamma \vdash e : \tau$ is provable.

Theorem 2 If $\Gamma \vdash e : \tau$ is provable, then $q(\Gamma) \vdash_p e : (\tau', \Gamma')$ is provable, where $close(\tau', \sigma, tv(\Gamma'))$, for some σ such that (σ, Γ') is the \preceq -smallest element of

$$\begin{array}{c}
\Gamma \vdash_{\text{p}} x : (\tau, \Gamma') \quad (\text{VAR}_{\text{p}}) \\
\text{where } (\tau, \Gamma') = \begin{cases} (\tau', \Gamma|_x) & \text{if } \Gamma(x) = \forall \alpha_i. \tau', \text{ for some } \{\alpha_i\}, i = 1..n, \tau' \\ & \text{where } \alpha_i \text{ are renamed to be fresh} \\ (\alpha, \{x : \alpha\}) & \text{otherwise, where } \alpha \text{ is a fresh type variable} \end{cases} \\
\frac{\Gamma \vdash_{\text{p}} e_1 : (\tau_1, \Gamma_1) \quad \Gamma, x : \sigma \vdash_{\text{p}} e_2 : (\tau_2, \Gamma_2)}{\Gamma \vdash_{\text{p}} \text{let } x = e_1 \text{ in } e_2 : (S\tau_2, S(\Gamma_2 \ominus x) \cup S\Gamma_1)} \quad (\text{LET}_{\text{p}}) \\
\text{where } \text{close}(\tau_1, \sigma, \text{tv}(\Gamma_1)), S = U(\Gamma_1, \Gamma_2) \\
\frac{\Gamma \ominus x \vdash_{\text{p}} e : (\tau, \Gamma_0)}{\Gamma \vdash_{\text{p}} \lambda x. e : (\tau' \rightarrow \tau, \Gamma')} \quad (\text{ABS}_{\text{p}}) \\
\text{where } (\tau', \Gamma') = \begin{cases} (\Gamma_0(x), \Gamma_0 \ominus x) & \text{if } x \in \text{dom}(\Gamma_0) \\ (\alpha, \Gamma_0) & \text{otherwise, where } \alpha \text{ is a fresh type variable} \end{cases} \\
\frac{\Gamma \vdash_{\text{p}} e_1 : (\tau_1, \Gamma_1) \quad \Gamma \vdash_{\text{p}} e_2 : (\tau_2, \Gamma_2)}{\Gamma \vdash_{\text{p}} e_1 e_2 : (S\alpha, S\Gamma_1 \cup S\Gamma_2)} \quad (\text{APPL}_{\text{p}}) \\
\text{where } S = \text{unify}(\mathcal{E}(\Gamma_1, \Gamma_2) \cup \{\tau_1 = \tau_2 \rightarrow \alpha\}) \text{ and } \alpha \text{ is a fresh type variable}
\end{array}$$

Figure 2: Type System ML_{p}

the set of solutions to $(e, q(\Gamma))$.

Theorem 3 For each e, Γ , there exists a unique (τ, Γ') , up to renaming of type variables, such that $\Gamma \vdash e : (\tau, \Gamma')$ is provable.

For typing problems of the form (e, Γ) such that Γ does not contain assumptions for variables in e that are required to have a quantified type, ML_{p} does not give any solution (and thus no principal solution, among the set of typing solutions derivable in ML); for example, the typing problem (xx, \emptyset) has infinitely many solutions in type system ML, but none is principal. For this problem, there is no solution in ML_{p} : there is no type σ such that $\emptyset \vdash_{\text{p}} xx : \sigma$ is derivable.

We present now a simple extension of ML_{p} , called ML_{o} , that derives a principal ML' typing for any given typing problem. Type system ML' is identical to ML except that it allows a typing context to have more than one assumption for any given variable (the type system is written without any modification at all). The definition of $(\Gamma, x : \sigma)$ is modified accordingly, as defined in section 1.

Any term e that is well-typed in type system ML (i.e. for which there are

Γ and σ for which $\Gamma \vdash e : \sigma$ is derivable) is also well-typed in type system ML' , and vice-versa (theorem 4).

Function lcg , defined in Figure 3, computes the least common generalisation for a set of simple types. A simplification is used, that considers lcg as a function by choosing any representative of the equivalence class of types τ that are least common generalisations of $\{\tau_i\}$ and differ only by renaming fresh type variables. Finite mappings are used in lcg' so that, for example, the least common generalisation of $\{\alpha_1 \rightarrow (\beta_1 \rightarrow \alpha_1), \alpha_2 \rightarrow (\beta_2 \rightarrow \alpha_2)\}$ is $\alpha \rightarrow (\beta \rightarrow \alpha)$, for some fresh type variables α, β (and not, say $\alpha \rightarrow (\beta \rightarrow \alpha')$).

$$\begin{aligned}
lcg(\{\sigma_i\}) &= lcg(\{\tau_i\}), \text{ where, for } i = 1, \dots, n, \\
&\sigma_i = \forall \alpha_{i_1} \dots \forall \alpha_{i_{m_i}}. \tau_i, \text{ for some } \alpha_{i_1}, \dots, \alpha_{i_{m_i}} \\
lcg(\{\tau_i\}) &= \tau, \text{ where } (\tau, m) = lcg'(\{\tau_i\}, \emptyset), \text{ for some } m \\
lcg'(\{\tau\}, m) &= (\tau, m) \\
lcg'(\{\tau_1, \tau_2\}, m) &= lcg_p(\{\tau_1, \tau_2\}, m) \\
lcg'(\{\tau_1, \tau_2\} \cup \mathcal{T}, m) &= lcg_p(\{\tau, \tau'\}, m') \\
\text{where } (\tau, m_0) &= lcg_p(\{\tau_1, \tau_2\}, m) \\
(\tau', m') &= lcg'(\mathcal{T}, m_0) \\
lcg_p(\{\alpha, \tau\}, m) &= (\text{ if } m(\alpha') = (\alpha, \tau), \text{ for some } \alpha', \text{ then } (\alpha', m) \\
&\text{ else } (\alpha', m \dagger \{\alpha' \mapsto (\alpha, \tau)\}), \text{ where } \alpha' \text{ is a fresh type variable}) \\
lcg_p(\{\tau_1 \rightarrow \tau_2, \tau'_1 \rightarrow \tau'_2\}, m) &= (\tau \rightarrow \tau', m') \\
\text{where } (\tau, m_0) &= lcg_p(\{\tau_1, \tau'_1\}, m) \\
(\tau', m') &= lcg_p(\{\tau_2, \tau'_2\}, m_0)
\end{aligned}$$

Figure 3: Least common generalisation of simple types

We define also $lcg(\Gamma) = \{x : \sigma \mid x \in dom(\Gamma) \text{ and } close(lcg(\Gamma(x)), \sigma, tv(\Gamma))\}$. Using \vdash' for type derivations in type system ML' , we have:

Theorem 4 $\Gamma \vdash e : \sigma$ is provable implies that $\Gamma \vdash' e : \sigma$ is provable, and $\Gamma \vdash' e : \sigma$ is provable implies that $lcg(\Gamma) \vdash e : \sigma$ is provable.

Derivations $\Gamma_0 \vdash_o e : (\tau, \Gamma)$ refer to derivations in ML_o . Function $unify_o$ is defined in Figure 4. It differs from the usual function $unify$ by the use of an additional boolean parameter. When set to true, the function simply ignores non-unifiable type equations; otherwise it behaves as usual. We define also:

$$U_o(\Gamma, \Gamma') = unify_o(\mathcal{E}(\Gamma, \Gamma'), true)$$

Type system ML_o is presented in Figure 5. As with ML' , type system ML_o also allows typing contexts to have more than one assumption for the

$$\text{unify}_o(\emptyset, b) = \emptyset$$

$$\begin{aligned} \text{unify}_o(E \cup \{\alpha = \tau\}, b) = \\ \text{if } \alpha \equiv \tau \text{ then } \text{unify}_o(E, b) \\ \text{else if } \alpha \text{ occurs in } \tau \text{ then if } b \text{ then } \text{unify}_o(E, b) \text{ else fail} \\ \text{else } \text{unify}_o(E[\tau/\alpha], b) \circ (\text{id} \uparrow (\alpha \mapsto \tau)) \end{aligned}$$

$$\text{unify}_o(E \cup \{\tau_1 \rightarrow \tau_2 = \tau'_1 \rightarrow \tau'_2\}, b) = \text{unify}_o(E \cup \{\tau_1 = \tau'_1, \tau_2 = \tau'_2\}, b)$$

Figure 4: Most general unifier, optionally ignoring non-unifiable equations

same variable. As an example, the principal typing solution to $(x x, \emptyset)$ is $(\alpha', \{x : \alpha, x : \alpha \rightarrow \alpha'\})$. The cardinality of $\Gamma(x)$ is used, in rule (ABS_o), to follow ML in allowing only simple types as types of lambda-bound variables. We have:

$$\Gamma_0 \vdash_o x : (\tau, \Gamma) \quad (\text{VAR}_o)$$

where $(\tau, \Gamma) = \text{if } \Gamma_0(x) \geq 1 \text{ then } (\text{lcg}(\Gamma_0(x)), \{x : \tau\})$

else $(\alpha, \{x : \alpha\})$, where α is a fresh type variable

$$\frac{\Gamma_0 \vdash_o e_1 : (\tau, \Gamma) \quad \Gamma_0, x : \sigma \vdash_o e_2 : (\tau', \Gamma')}{\Gamma_0 \vdash_o \text{let } x = e_1 \text{ in } e_2 : (S\tau', S(\Gamma' \ominus x) \cup S\Gamma)} \quad (\text{LET}_o)$$

where $\text{close}(\tau, \sigma, \Gamma)$, $S = U_0(\Gamma, \Gamma')$

$$\frac{\Gamma_0 \ominus x \vdash_o e : (\tau, \Gamma)}{\Gamma_0 \vdash \lambda x. e : (\tau' \rightarrow \tau, \Gamma')} \quad \#\Gamma(x) \leq 1 \quad (\text{ABS}_o)$$

where $(\tau', \Gamma') = \text{if } x : \tau_0 \in \Gamma$, for some τ_0 then $(\tau_0, \Gamma - \{x : \tau_0\})$

else (α, Γ) , where α is a fresh type variable

$$\frac{\Gamma_0 \vdash e_1 : (\tau_1, \Gamma_1) \quad \Gamma_0 \vdash e_2 : (\tau_2, \Gamma_2)}{\Gamma_0 \vdash e_1 e_2 : (S\alpha, \Gamma)} \quad (\text{APPL}_o)$$

where $S_0 = \text{unify}(\{\tau_1 = (\tau_2 \rightarrow \alpha)\})$, $S_1 = U_o(S_0\Gamma_1, S_0\Gamma_2)$

$S = S_1 \circ S_0$, $\Gamma = S\Gamma_1 \cup S\Gamma_2$, α is a fresh type variable

Figure 5: Type System ML_o

Theorem 5 If $\Gamma_0 \vdash_o e : (\tau, \Gamma)$ is provable, then $\Gamma \vdash' e : \tau$ is provable.

Theorem 6 If $\Gamma \vdash' e : \tau$ is provable, then, for any set of term variables X , $\Gamma \ominus X \vdash_o e : (\tau', \Gamma')$ is provable, where, letting σ be such that $close(\tau', \sigma, \Gamma')$, (σ, Γ') is the \preceq -smallest element of the set of solutions to $(e, \Gamma \ominus X)$.

5 Conclusion

In this paper we show that there is a type system for core-ML that, using exactly the same syntax of types of the Damas-Milner system, types exactly the same terms of this system, and has principal typings.

A definition of principal typing is given, capturing the basic, simple idea of representing the set of all typings that can be obtained in derivations for the relevant term in a given type system. This definition is parameterised on an ordering on types, enabling it to be used for different type systems.

A type system for core-ML is presented that uses type expressions with exactly the same form as the Damas-Milner system, and considers as well-typed exactly the same expressions of the Damas-Milner system. A type inference algorithm is then presented, which computes principal typings with respect to the given type system.

References

- [AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
- [CDDK86] D. Clement, D. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative Language Mini-ML. In *ACM Symp. on List and Functional Programming*, pages 13–27, 1986.
- [CF58] H.B. Curry and R. Feys. *Combinatory Logic I*. North-Holland, 1958.
- [Chu40] Alonzo Church. A formulation of the simple theory of types. *J. Symbolic Logic*, (5):56–58, 1940.
- [CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4), December 1985.
- [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.
- [FM90] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, 1990.

- [Gir71] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *2nd Scandinavian Logic Symposium*, pages 63–92, 1971.
- [Gir72] Jean-Yves Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. These D'Etat, 1972.
- [Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM TOPLAS*, 15(2):253–289, Apr 1993.
- [Hin69] J.R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. AMS*, 146:29–60, 1969.
- [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.
- [MH88] John Mitchell and Robert Harper. The essence of ML. *Proc. 5th ACM Symp. on Principles of Programming Languages*, pages 28–46, 1988.
- [MH93] John Mitchell and Robert Harper. On the type structure of standard ml. *ACM TOPLAS*, 15(2):211–252, Apr 1993.
- [Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
- [Mit88] John Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2/3):211–249, 1988.
- [Mit96] John Mitchell. *Foundations for programming languages*. MIT Press, 1996.
- [MP88] John Mitchell and Gordon Plotkin. Abstract Types Have Existential Type. *ACM Transactions on Programming Languages and Systems*, 1988.
- [Rey74] John Reynolds. Towards a theory of type structure. In *Paris Colloq. on Programming*, pages 408–425, 1974. Springer-Verlag LNCS 19.