Type Inference with Rank 1 Polymorphism for Type-Directed Compilation of ML*

Atsushi Ohori Nobuaki Yoshida

Research Institute for Mathematical Sciences
Kyoto University, Kyoto 606-8502, Japan
{ohori,nyoshi}@kurims.kyoto-u.ac.jp

Abstract

This paper defines an extended polymorphic type system for an ML-style programming language, and develops a sound and complete type inference algorithm. Different from the conventional ML type discipline, the proposed type system allows full *rank 1 polymorphism*, where polymorphic types can appear in other types such as product types, disjoint union types and the range types of function types. Because of this feature, the proposed type system significantly reduces the "value-only" restriction of polymorphism, which is currently adopted in most of ML-style impure languages. It also serves as a basis for efficient implementation of type-directed compilation of polymorphism. The extended type system achieves more efficient type inference algorithm, and it also contributes to develop more efficient type-passing implementation of polymorphism. We show that the conventional ML polymorphism sometimes introduces exponential overhead both at compile-time elaboration and run-time type-passing execution, and that these problems can be eliminated by our type inference system. The proposed type inference algorithm only uses the conventional first-order unification, and is easily adopted in existing implementation of ML family of languages.

1 Introduction

ML type discipline [14, 1] achieves both the flexibility of programming through ML's polymorphic let construct, and practical type inference through the restricted treatment of polymorphism. Compared with the full second-order type discipline [3, 22], ML only allows type abstraction at top level. Due to this restriction, any typable ML program has a principal type, which can be computed by a unification-based type inference algorithm. These simple properties make ML type inference system particularly suitable for programming language implementation. Despite the existence of more powerful polymorphic type inference systems such as [10, 12], ML type inference system appears to be the most practical one and it has been widely used in a number of actual programming languages. As we shall explain below, however, the current ML type system and the type inference algorithm exhibit serious limitations and problems in design and implementation of a practical polymorphic programming language, especially in connection with *value polymorphism* and *type-passing implementation* of polymorphism in recently emerging type-directed compilation. The motivation of our work of extending ML type discipline is not to provide more expressiveness but to solve those problems and to provide a basis for a better design and implementation of an ML-style language. Let us review the problems related to value polymorphism and type passing semantics in ML.

1.1 Problem in value polymorphism

To safely integrate imperative features, in most of currently implemented ML-style impure polymorphic languages including Standard ML [15] and Ocaml [13], polymorphism is restricted to syntactic "values" (non expansive expressions). As argued in [27], this is a simple and easily implementable solution to the subtle problem of the inconsistency between polymorphism and imperative features. However, the combination of ML polymorphism and the value restriction results in a gross over restriction, excluding a number of safe programs.

For very simple examples, consider the following two functions (written in SML syntax):

```
val f = ((fn x => x) 1 ,
         fn x => fn y => (x + 1,ref y))
val g = (#2 f) 1
```

In both cases the source of polymorphism is due to the values part in the programs, and therefore both are safely given a polymorphic type. In the first case, the only polymorphic part is the second component of the function `fn x => fn y => (x,ref y)` which can safely be given a polymorphic type `'b -> 'a -> 'b * 'a ref`. The second involes two computation steps. Since the projection `(#2 f)` simply return the second component from a pair of value and therefore the result type is the same as that of the seocnd component of f. The application of the result of the projection to a `1` may involve arbitrary computation. However, since this application only imvolve monomorphic computation, the result can be given a polymorphic type `'a -> int * 'a ref`, which corresponds to value `fn y => (x,ref y)` with `x` bound to 1.

*This is the authors' version of the paper published in ACM International Conference on Functional Programming (ICFP), pages 160 – 171 1999.

Unfortunately, however, both of them (and many other similar programs) are rejected by the current ML type system because of the restricted treatment of ML polymorphism. Under the ML type discipline, value restriction implies that any data structure containing non-value part cannot be polymorphic even those the "non-value part" is monomorphic. We find this restriction unreasonable. This situation is particularly unfortunate when a modern language such as Standard ML provide a variety of rich data structures which can freely contain higher-order objects.

1.2 Problem in type-directed compilation

Another major motivation of this work comes from implementaqtion of type-directed specialization of polymorphism [19]. In this paradigm, the static type information is used at runtime to achieve efficient implementation of polymorphic functions. This approach has been proposed for compilation of polymorphic records [18], and a number of similar methods have been developed including overloading resolution [21], intentional type analysis [6], and unboxed operational semantics [20]. Tag-free garbage collection [26] also uses type information at run-time and requires a similar run-time structure.

In these approaches, type abstraction is compiled to abstraction on type information and type application is compiled to application. Under this implementation strategy, however, ML's restricted polymorphism sometimes introduces unacceptable runtime overhead. Since ML polymorphism only allows type abstraction at top-level, every time when polymorphic functions are used to define another polymorphic functions, polymorphic instantiation and polymorphic generalization are performed. Under type-directed compilation, this implies the accumulation of runtime overhead.

To understand the problem, let us consider the following simple progam in a polymorphic record calculus [18].

```
fun xval {X=x,...} = x ;
fun yval {Y=y,...} = y ;
val methods = {getX = xval, getY = yval};
val point1 = {Methods=methods, X = 1, Y = 2};
```

The function `xval` is given the polymorphic type $\forall t_1.\forall t_2 :: \{\!\{X : t_1\}\!\}.t_2 \to t_1$ where type abstraction $\forall t_2 :: \{\!\{X : t_1\}\!\}$ indicates that t_2 is a record type containing the filed $X : t_1$. This type abstraction is compiled to a lambda abstraction receiving the index value corresponding to the `X` filed in the argument record, and an appropriate application is inserted when this type is instantiated. Thus the above program is compiled to the following code.

```
val f = fn I => fn x => x[I];
val g = fn I => fn x => x[I];
val methods = fn I1 => fn I2 =>
    {getX = f I1, getY = g I2};
val point1 = fn I3 => fn I4 =>
    {Methods=methods I3 I4, X = 1, Y = 2};
```

As seen from this example, due to ML's restricted polymorphism, abstractions and applications are accumulated every time a polymorphic function is used to build another polymorphic function.

This problem is inherent in all the other appoaches that uses type information at run-time based on type-passing implementation mentioned above. In this paper, instead of considering any particular primitive that use type information at run-time, we simply assume that type abstraction and type application are compiled to lambda abstruction and lambda application to pass type information at runtime, and we concentrate on the core ML terms with products and disjoint union.

Figure 1 shows an example SML program and the corresponding explicitly typed term. Execution of the compiled code takes the time exponential in the size of the program under type-passing implementation, while the conventional implementation takes linear time. In this example, the numer of type varibles is exponential, but this is not essential. The following code uses linear number of type variables, but its still takes exponential time under the type passing semantics.

```
let
   val x = [fn x => x, fn x => x]
   val x = [x,x]
   :
   val x = [x,x]
in
   (hd (··· (hd x)···)) 1
end
```

In a real program, such an extreme case may not often occur. But at least, runtime overhead increases in proportion to the number of nested type abstractions and type applications. In a large and modular program, this may result in a significant reduction of the runtime efficiency. It may be the case that for simple cases, type passing overhead can be eliminated by some ad-hoc optimization methods. When code become complicated, however, we expect that such optimization is quite difficult if not impossible. It should also be noted that this overhead is not something that can be eliminated by optimization of runtime type representation or type passing mechanism such as those studied in [16, 23].

1.3 Solution by Runk 1 Type Inference

The problems in both cases can be eliminated if ML type system is extended so that data structure can contain polymorphic programs having a polymorphic type.

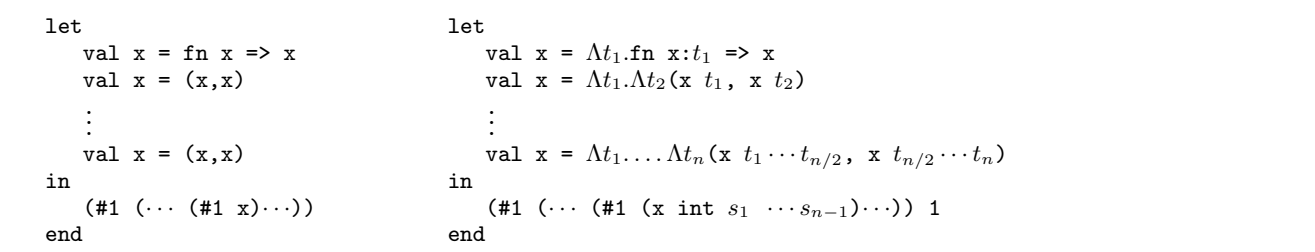
If type generalization can be performed on each component in a compound data structure, then the scope of type generalization is more likely to be limited to a function definition and therefore significantly more programs can be accepted under value polymoprhism. For example, in such an extended type system, the following explicitly typed lambda terms can be constructed for the functions `f` and `g` given in the beggining of this paper.

```
val f = ((fn x:int => x) 1 ,
         fn x:int => Λt.fn y:t => (x + 1,ref y))
       :  int * (int → (∀t.t → int * ref(t)))

val g = (#2  g) int 1
       :  ∀t.t → int * ref(t)
```

where t is a type variable. Λt is type abstraction and $ref(t)$ is a reference type whose component type is t. Since type abstractions are performed on values, both terms satisfy the value polymorphism restriction.

Such an extended type system will also reduce the runtime type-passing overhead by eliminating intermediate type



Spoure Code Corresponding explicitly typed term

Figure 1: Example SML code that exhibits exponential overhead both in compile-time and run-time

applications and type abstractions. For example, the following explicitly typed term can be constructed for the example code in Figure 1.

```
let
   val x = Λt.fn x:t =>x
   val x = (x,x)
   ⋮
   val x = (x,x)
in
   (#1 (··· (#1 x)···)) int 1
end
```

The type-passing overhead in this program is a small constant; the program performs only one type abstraction and one type instantiation. This is in sharp contrast with exponential run-time overhead under the conventional ML type system. From this example, it is also expected that type inference in the extended type system can be more efficient by avoiding intermediate type instantiation and type abstraction. As we shall see later, this is indeed the case. For example, type inference for the above code takes exponential time in the conventional ML type system, while it only takes linear time in our type inference algorithm developed below.

The goal of this paper is to define an extended type system that is strong enough to have those features and to develop a practical unification based type inference algorithm for the extended type system.

For our purpose, it appears to be sufficient to extend the set of ML polymorphic types to be those types where polymorphic types can occur at any *strictly positive* positions in other type constructors (i.e. those that are not at the left of any function arrow.) Under the definition of [10], the set of types is characterized as the set $R(1)$ of rank 1 polymorphic types (extended with products and disjoint unions). For the completeness of presentation, we repeat the definition of rank k polymorphic types given in [10] below.

$$R(k) = R(k-1) \mid \forall \bar{t}.R(k) \mid R(k-1) \to R(k)$$

The major technical contribution of the present paper is to establish a practical type inference system for ML with full rank 1 types. Specifically, we carry out the following.

1. We define an ML-style type system with products and disjoint unions extended with rank 1 polymorphism, which we call $\mathrm{ML}^{\mathrm{R}(1)}$.
2. We prove that for a raw term without disjoint unions, it is typable in ML if and only if it is typable in the extended type system. Rank 1 polymorphic type system is however strictly more powerful than that of ML for terms involving disjoint unions.
3. We give a type inference algorithm $\mathcal{W}^{R(1)}$ in the style of algorithm $\mathcal{W}$ and prove that it is sound and complete with respect to the extended type system.

The item 2 may require some explanation. This property says that as far as the typability is concerned, the extended type system is almost equivalent to that of ML. This equivalence is modulo a strong equivalence relation on polymorphic types we shall define, which collapses the runtime effects in type-passing semantics. In ML-style implicit type discipline, type systems of equivalent typability may exhibit quite different runtime behavior under type-passing semantics. This is analogous to the fact that there are various programs having the same denotation but having different runtime complexity. Our claim is that our type inference system is indeed significantly better than the current ML type discipline for type-directed compilation realizing type-passing semantics.

These results can be readily transferred to existing implementation of ML-style polymorphic languages. The authors have implemented an algorithm for type inference and reconstruction of explicitly typed terms. After we have presented the type inference algorithm, we show the actual output of the inferred type and the explicitly typed term computed by our prototype system. We are now implementing a prototype type-directed compiler for ML with record polymorphism incorporating the result presented in this paper.

1.4 Comparison to related works

Before giving the technical development, we compare our work with existing related works.

As far as the typability is concerned, the solvability of type inference problem has been already established for rank 2 polymorphism [10, 12], which properly contains our type system. However, these results are either indirect [10] or involves order constraint due to semi-unification [12]. A more direct algorithm is given in [8], but it still involves order constraints on types. There are also several studies on extended polymorphic type inference algorithms based on semi-unification [7, 11]. In a theoretical perspective, it can be argued that type inference with rank 2 polymorphism is no more complicated than those of ML. Indeed, ML type inference is intractable in the worst case [9] and the complexity of its typability is polynomial-time equivalent to much more

powerful rank 2 polymorphic type system [10]. In a practical language design, however, there appear to be qualitative difference in understanding a type and in presenting a type to the user.

To our knowledge, there is no known unification based direct type inference algorithm for rank 1 polymorphism. The type inference algorithm presented here will contribute directly to improve existing practical programming language implementation. Although certain amount of delicate technical development is required to establish its completeness, the type inference algorithm itself requires surprisingly little modification to algorithm $\mathcal{W}$ as presented in [1], and therefore can be readily incorporated in existing implementations. The necessary mechanisms for value restriction and type-passing implementation are the same as those for the conventional ML type system. Moreover, inferred type information is no more complex than those of ML and is easily understabable.

1.5 Organization of the paper

The rest of the paper is organized as follows. Section 2 defines the language, $\mathrm{ML}^{\mathrm{R}(1)}$, with rank 1 polymorphism. Section 3 compares the type system of $\mathrm{ML}^{\mathrm{R}(1)}$ with that of ML, and shows that for the fragment without disjoint union, the typability is equivalent. Section 4 gives the type inference algorithm and proves its soundness and completeness. Section 5 discusses several issues in incorporating the presented type system and the type inference algorithm in an actual ML compiler.

2 The $\mathrm{ML}^{\mathrm{R}(1)}$ Type System

We consider the following set of raw ML terms with products and disjoint unions.

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.e \mid e\ e \mid \\ & & (e,e) \mid e.1 \mid e.2 \mid inj_2(e) \mid inj_1(e) \mid \\ & & case\ e\ of\ inj_1(x) \Rightarrow e, inj_2(x) \Rightarrow e \mid \\ & & let\ x = e_1\ in\ e_2 \end{array}$$

(e_1, e_2) is a pair, $e.1, e.2$ are the first and the second projection, respectively. $inj_1(e), inj_2(e)$ are left and right injections to a disjoint union, respectively.

We let t range over a given countably infinite set of *type variables*, and let T range over finite sets of type variables. Following Damas and Milner's [1] type system of ML, we divide the set of types into the set of *monotypes* (ranged over by τ) and *polytypes* (ranged over by σ).

$$\begin{array}{rcl} \tau & ::= & t \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \\ \sigma & ::= & \tau \mid \forall T.\sigma \mid \tau \to \sigma \mid \sigma \times \sigma \mid \sigma + \sigma \end{array}$$

These sets are generalization of $R(0)$ (rank 0 polymorphic types) and $R(1)$ (rank 1 polymorphic types) with product and disjoint union types, respectively.

A *type substitution*, or simply *substitution*, is a function from a finite set of type variables to monotypes. We write $[\tau_1/t_1, \ldots, \tau_n/t_n]$ for the substitution that maps each t_i to τ_i. A substitution S is extended to the set of all type variables by letting $S(t) = t$ for all $t \notin dom(S)$, and it in turn is extended uniquely to monotypes. The result $S(\tau)$ of applying substitution S to a monotype τ is understood as the result of applying the extension of S to τ. The *composition* of two substitution S_1, S_2, written as S_2S_1, is defined as a substitution S' such that $dom(S') = dom(S_1)$, and for any $t \in dom(S_1)$, $S_2(S_1(t))$ where S_2 is regarded as the unique extension of S_2. We assume that composition is associated to the right and write $S_1S_2 \cdots S_{n-1}S_n$ for $S_1(S_2(\cdots(S_{n-1}S_n)))$. The result of applying a substitution S to a second-order type $\forall T.\sigma$ is the type obtained by applying S to its all *free type variables*. Under the bound type variable convention, we can simply take $S(\forall T.\sigma) = \forall T.S(\sigma)$.

Polytypes are considered modulo the equivalence relation induced by the following rules:

$$\begin{array}{rcl} \forall \emptyset.\sigma & = & \sigma \\ \forall T_1.(\forall T_2.\sigma) & = & \forall T_1 \cup T_2.\sigma \\ \forall T_1 \cup T_2.\sigma & = & \forall T_1.\sigma \ (\text{ if } T_2 \cap FTV(\sigma) = \emptyset) \end{array}$$

To distinguish this syntactic equivalence relation from other equivalence relations defined later, we call this relation *α-equivalence on polytypes*. Under this equivalence, we can assume the following convention on bound variables of polytypes: (1) all bound type variables are distinct and different from any free type variables, and (2) any polytype $\forall T.\sigma$ satisfies $T \subseteq FTV(\sigma)$. The first condition is the usual "bound variable convention" for the bound type variables.

In order to define a type system for $\mathrm{ML}^{\mathrm{R}(1)}$ we need to define a polymorphic instantiation relation. Instantiation involves substitution of type variables for bound type variables. Because of this, we need to work on representatives of equivalence classes of polytypes. To make our presentation simpler, in the following definition of instantiation, we implicitly assume that a polytype is a unique canonical representation of its equivalence class. It is easy to come up with a unique representation. Below, we give one possible definition we used in our implementation.

We extend the set of type variables with natural numbers and use them for bound type variables in canonical representations. We write (i, j) for the set $\{i, \ldots, i + j\}$. Let $\forall T.\sigma$ be a given polytype satifying the bound variable convention. We first recursively compute a canonical representation $\overline{\sigma}$ of σ. T is linearly ordered with respect to $\overline{\sigma}$ in such a way that $t_1 <_\sigma t_2$ iff there is an occurrence of t_1 whose tree address is smaller (in lexicographical ordering on tree addresses) than the address of any occurrence of t_2. (See [4] for the details of tree address.) Let $\{t_1, \ldots, t_n\} = T$ such that $t_i <'_\sigma t_j$ if $i < j$. Let m be the maximal natural number used as bound variables in $\overline{\sigma}$. The representation $\overline{\forall T.\sigma}$ of $\forall T.\sigma$ is then defined as follows.

$$\overline{\forall T.\sigma} = \forall(m+1, m+n).[m+1/t_1, \ldots, m+n/t_n]\overline{\sigma}$$

The folloing is an example.

$$\begin{array}{l} \overline{\forall\{s,t\}.t \to (\forall\{t,u\}.t \to s \times u)} \\ \quad = \ \forall(3,2).3 \to (\forall(1,2).1 \to 4 \times 2) \end{array}$$

Figure 2 gives a recursive algorithm to compute the canonical representation $\overline{\sigma}$ of σ.

We turn to the definition of instantiation under this implicit assumption. Since polytypes can appear in substructures of a given type, instantiation need to be defined *structurally*. For this purpose, we introduce a syntactic category of "instantiations". An *instantiation* (ranged over by $\mathcal{I}$) is a composite structure made up with substitution given by the following syntax:

$$\mathcal{I} ::= * \mid \forall S.\mathcal{I} \mid * \to \mathcal{I} \mid \mathcal{I} \times \mathcal{I} \mid \mathcal{I} + \mathcal{I}$$

$$
\begin{array}{rcl}
R(i, \forall T.\sigma) & = & \mathit{let}\ (j,\sigma') = R(i,\sigma) \\
& & \quad \{t_1,\ldots,t_m\} = T'\ (t_i <_{\sigma'} t_j \text{ if } i<j) \\
& & \quad \sigma'' = [j+1/t_1,\ldots,j+m/t_m]\sigma' \\
& & \mathit{in}\ (j+m, \forall(j+1, j+m).\sigma'') \\
R(i, \tau\to\sigma) & = & \mathit{let}\ (j,\sigma') = R(i,\sigma)\ \mathit{in}\ \tau\to\sigma' \\
R(i, \sigma_1\times\sigma_2) & = & \mathit{let}\ (j,\sigma_1') = R(i,\sigma_1) \\
& & \quad (j',\sigma_2') = R(j,\sigma_2) \\
& & \mathit{in}\ \sigma_1'\times\sigma_2' \\
R(i, \sigma_1+\sigma_2) & = & \mathit{let}\ (j,\sigma_1') = R(i,\sigma_1) \\
& & \quad (j',\sigma_2') = R(j,\sigma_2) \\
& & \mathit{in}\ \sigma_1'+\sigma_2' \\
\\
\overline{\sigma} & = & \mathit{let}\ (n,\sigma') = R(0,\sigma)\ \mathit{in}\ \sigma'
\end{array}
$$

Figure 2: Canonical represetation of polytypes

where $*$ is the empty instantiation and $\forall S.\mathcal{I}$ is one that performs instantiation for a polytype of the form $\forall T.\sigma$ such that $dom(S) = T$. To define applicability of instantiation to polytypes, we introduce a simple kind system for polytypes and instantiations. The set of kinds (ranged over by k) is defined by the following grammar.

$$k ::= * \mid \forall T.k \mid * \to k \mid k \times k \mid k + k$$

The kinding relation $\sigma :: k$ is given below.

$$\vdash \tau :: * \qquad \frac{\vdash \sigma :: k}{\vdash \forall T.\sigma :: \forall T.k}$$

$$\frac{\vdash \sigma :: k}{\vdash \tau \to \sigma :: * \to k} \qquad \frac{\vdash \sigma_1 :: k_1 \quad \vdash \sigma_2 :: k_2}{\vdash \sigma_1\times\sigma_2 :: k_1 \times k_2}$$

$$\frac{\vdash \sigma_1 :: k_1 \quad \vdash \sigma_2 :: k_2}{\vdash \sigma_1+\sigma_2 :: k_1 + k_2}$$

The kinding on instantiation $\mathcal{I} :: k$ is defined analogously. Below, we only show the cases for $*$ and $\forall S.\mathcal{I}$.

$$\vdash * :: * \qquad \frac{\vdash \mathcal{I} :: k \quad dom(S) = T}{\vdash \forall S.\mathcal{I} :: \forall T.k}$$

For a polytype σ and an instantiation $\mathcal{I}$ having the same kind, the *instance of σ under $\mathcal{I}$*, denoted by $\mathit{Inst}(\sigma,\mathcal{I})$, is defined as follows.

$$
\begin{array}{rcl}
\mathit{Inst}(\tau, *) & = & \tau \\
\mathit{Inst}(\forall T.\sigma, \forall S.\mathcal{I}) & = & \mathit{Inst}(S(\sigma),\mathcal{I}) \\
\mathit{Inst}(\tau_0\to\sigma, *\to\mathcal{I}) & = & \tau_0 \to \mathit{Inst}(\sigma,\mathcal{I}) \\
\mathit{Inst}(\sigma_1\times\sigma_2, \mathcal{I}_1\times\mathcal{I}_2) & = & \mathit{Inst}(\sigma_1,\mathcal{I}_1)\times\mathit{Inst}(\sigma_2,\mathcal{I}_2) \\
\mathit{Inst}(\sigma_1+\sigma_2, \mathcal{I}_1+\mathcal{I}_2) & = & \mathit{Inst}(\sigma_1,\mathcal{I}_1)+\mathit{Inst}(\sigma_2,\mathcal{I}_2)
\end{array}
$$

It is easily verified that this operation is well defined for any pair of a polytype and an instantiation having the same kind. The following is a simple example.

$$
\begin{array}{rcl}
& & \mathit{Inst}(\mathit{int}\to(\forall\{t_1,t_2\}.t_1\to t_2)\times(\forall\{t_3\}.t_3+t_3), \\
& & \quad *\to(\forall[\mathit{int}/t_1,\mathit{bool}/t_2].*)\times(\forall[t_4/t_3].*)) \\
& = & \mathit{int}\to((\mathit{int}\to\mathit{bool})\times(t_4+t_4))
\end{array}
$$

The ordering on polytypes, denoted by $\sigma_1 \le \sigma_2$, is then give as follows.

$$\sigma_1 \le \sigma_2 \Longleftrightarrow \exists\mathcal{I}.\sigma_1 = \mathit{Inst}(\sigma_2,\mathcal{I})$$

Using these definitions and notations, the type system of $\mathrm{ML}^{\mathrm{R}(1)}$ is given in Figure 3 as a proof system to derive a *typing judgement* of the form $\Gamma \rhd e : \sigma$ indicating the fact that e has type σ under type assignmnet Γ (which is a function from a finite subset of variables to polytypes.) In this type system, there are three cases where types are restricted to be monotypes: the argument type of functions in rules (ABS) and (APP), and the result type of case branches in rule (CASE). These are exactly the cases where unification are called for in the ML-style type inference algorithm. By restricting those to be monotypes, the type system allows a unification based type inference algorithm, as we shall show in detail later.

In rule (CASE), type abstraction is allowed for σ_1 and σ_2 independently at the time of binding to variables. Without this extra generality, it is rather difficult to design a type inference algorithm that is sound and complete with respect to the type system. The reason is the following. Suppose the rule (CASE) does not allow type generalization. For a type inference algorithm to be sound, the algorithm must assume the same σ_1 and σ_2 of $\sigma_1+\sigma_2$ which is inferred for e_1. On the other hand, for a type inference algorithm to be complete, σ_1 and σ_2 must be the maximal ones with respect to the ordeing $\le$. In a unification based type inference system, however, componets of an inferred type are usually not maximal respect to the ordeing $\le$. Note that the rule (TABS) is not sufficient to achive maximality in σ_1 and σ_2 separeyely, since this rule only generalzes an entire type. Adding this generality preserves semantic soundness. One way to verify this is to note that $\forall T.\sigma_1+\sigma_2$ is isomorphic to $(\forall T.\sigma_1)+(\forall T.\sigma_2)$ in the sense of [2].

The typing relation is stable under substitution, as shown in the following.

Lemma 1 *If* $\mathrm{ML}^{\mathrm{R}(1)} \vdash \Gamma \rhd e : \sigma$ *then* $\mathrm{ML}^{\mathrm{R}(1)} \vdash S(\Gamma) \rhd e : S(\sigma)$.

To establish various properties of this type system later, we define an equivalence relation and a generic ordering on polytypes with respect to a given context. We define a *free type variable context* to be a set of free type variables, which are those type variables that may appear free in other types such as those in type assignment. Due to the implicit nature of ML, a polytype may be equivalent to monotypes containing free type variables that do not appear in its context. For this reason, we need to define type equivalence relative to a free type variables context C. We write

$$C \vdash \sigma_1 \cong \sigma_2$$

to denote the fact that σ_1 is equivalent to σ_2 with respect to C. This relation is defined in two stages. We first define the relation $\sigma_1 \cong \sigma_2$. Let F range over contexts of type expression given by the grammar

$$F = [\,] \mid \tau\to F \mid F\times\sigma \mid \sigma\times F \mid F+\sigma \mid \sigma+F$$

where $[\,]$ denotes a "hole". We write $F[\sigma]$ for the type obtained by filling the hole of F with σ. The relation $\sigma \cong$

$$(\text{VAR})\quad \Gamma\{x:\sigma\} \rhd x : \sigma$$

$$(\text{ABS})\quad \frac{\Gamma\{x:\tau\} \rhd e : \sigma}{\Gamma \rhd \lambda x.e : \tau \to \sigma} \qquad (\text{APP})\quad \frac{\Gamma \rhd e_1 : \tau \to \sigma \quad \Gamma \rhd e_2 : \tau}{\Gamma \rhd e_1\ e_2 : \sigma}$$

$$(\text{PROD})\quad \frac{\Gamma \rhd e_1 : \sigma_1 \quad \Gamma \rhd e_2 : \sigma_2}{\Gamma \rhd (e_1, e_2) : \sigma_1 \times \sigma_2} \qquad (\text{PROJ}_i)\quad \frac{\Gamma \rhd e : \sigma_1 \times \sigma_2}{\Gamma \rhd e.i : \sigma_i} \quad i \in \{1,2\}$$

$$(\text{INJ}_i)\quad \frac{\Gamma \rhd e : \sigma_i}{\Gamma \rhd inj_i(e) : \sigma_1 + \sigma_2} \quad i \in \{1,2\}$$

$$(\text{CASE})\quad \frac{\Gamma \rhd e_1 : \sigma_1 + \sigma_2 \quad \Gamma\{x : \forall T_1.\sigma_1\} \rhd e_2 : \tau \quad \Gamma\{y : \forall T_2.\sigma_2\} \rhd e_3 : \tau}{\Gamma \rhd case\ e_1\ of\ inj_1(x) \Rightarrow e_2, inj_2(y) \Rightarrow e_3 : \tau} \quad (T_1 \cup T_2) \cap FTV(\Gamma) = \emptyset$$

$$(\text{TABS})\quad \frac{\Gamma \rhd e : \sigma}{\Gamma \rhd e : \forall T.\sigma} \quad \text{if } T \cap FTV(\Gamma) = \emptyset \qquad (\text{TAPP})\quad \frac{\Gamma \rhd e : \sigma_1}{\Gamma \rhd e : \sigma_2} \quad \text{if } \sigma_2 \le \sigma_1$$

$$(\text{LET})\quad \frac{\Gamma \rhd e_1 : \sigma_1 \quad \Gamma\{x : \sigma_1\} \rhd e_2 : \sigma_2}{\Gamma \rhd let\ x = e_1\ in\ e_2 : \sigma_2}$$

Figure 3: $\text{ML}^{\text{R}(1)}$ Type System

σ' is the reflexive, symmetric and transitive closure of the following axiom schemes.

$$\forall T_1.F[\forall T_2.\sigma] \cong \forall T_1 \cup T_2.F[\sigma]$$

where we assume bound type variable convention. Any $\text{ML}^{\text{R}(1)}$ polytype is equivalent to a unique ML polytype (modulo α-equivalence of polytypes defined earlier.) For a given $\text{ML}^{\text{R}(1)}$ polytype σ, we write $[\sigma]_{\cong}$ for the ML polytype equivalent to σ. Below is a simple example.

$$[\forall t_1.t_1 \to \forall t_2.t_2 \times t_1]_{\cong} = \forall\{t_1,t_2\}.t_1 \to (t_2 \times t_1)$$

For a free type variable context C, we use the notation $Clos(\sigma, C)$ for the polytype $\forall T.\sigma$ such that $T = FTV(\sigma) \setminus FTV(C)$. The equivalence relation with respect to a context is then defined as follows.

$$C \vdash \sigma_1 \cong \sigma_2 \iff Clos(\sigma_1, C) \cong Clos(\sigma_2, C)$$

This equivalence relation says that any type is equivalent to a polytype obtained by quantifying free type variables that do not occur in the free type variable context, and it in turn is equivalent to the polytype obtained by moving all the type quantifiers at the top level. For example,

$$\{t_1,t_2\} \vdash t_1 \to t_2 \times (t_3 \to t_3) \cong t_1 \to t_2 \times (\forall t_3.t_3 \to t_3)$$

For a given C and σ there is a unique (upto α-equivalence) σ' satisfying the following: $C \vdash \sigma \cong \sigma'$, σ' is an ML polytypem and $FTV(\sigma') \setminus C = \emptyset$. We write $[\sigma]^C_{\cong}$ for σ' satisfying the above condition. Below is a simple example.

$$[t_1 \to t_2 \times (t_3 \to t_3)]^{\{t_1,t_2\}}_{\cong} = \forall t_3.t_1 \to t_2 \times (t_3 \to t_3)$$

Using these, we define the ordering relation on polytypes as follows.

$$\begin{aligned} \sigma_1 \preceq \sigma_2 &\iff [\sigma_1]_{\cong} \le [\sigma_2]_{\cong} \\ C \vdash \sigma_1 \preceq \sigma_2 &\iff [\sigma_1]^C_{\cong} \le [\sigma_2]^C_{\cong} \end{aligned}$$

Defining the ordering on polytypes modulo equivalence and relative to the set of free type variables in the context is essential in establishing the correspondence between the type system of $\text{ML}^{\text{R}(1)}$ and that of ML. It should be noted that the necessity of the latter is already seen in establishing the completeness of ML type inference. The ordering relation on ML polytypes used in Damas-Milner[1] type system, denoted here by $\preceq_{ML}$, is characterized as

$$\sigma_1 \preceq_{ML} \sigma_2 \iff FTV(\sigma_2) \vdash \sigma_1 \preceq \sigma_2$$

For the convenience of the following development, we use the notation: $freshInst(\sigma)$ for the monotype obtained from σ by replacing each bound type variable in σ with a distinct fresh type variable. Here "fresh type variables" mean those that do not appear in σ and its surrounding context. For any given C and σ, it is always the case that $C \vdash \sigma \cong freshInst(\sigma)$.

3 Relationship between $\text{ML}^{\text{R}(1)}$ and ML Type Systems

Although the type system defined above is more general than that of ML, for the expressions not involving disjoint union types, the typability is shown to be equivalent. For those expressions involving disjoint union types, however, $\text{ML}^{\text{R}(1)}$ type system is strictly more powerful. This is because, in $\text{ML}^{\text{R}(1)}$, variable binding in case branches is polymorphic.

Damas-Milner type system for ML is given in Figure 4. Since $\text{ML}^{\text{R}(1)}$ type system is an extension of that of ML, it is immediate that any ML derivation is also a derivation of $\text{ML}^{\text{R}(1)}$. We show that for expressions not involving disjoint union types, the converse also holds under our interpretation of type equivalence and polytype ordering. We write $[\Gamma]_{\cong}$ for the type assignment Γ' such that $dom(\Gamma') = dom(\Gamma)$ and $\forall x \in dom(\Gamma).\Gamma'(x) = [\Gamma(x)]_{\cong}$.

Theorem 2 *If* $\text{ML}^{\text{R}(1)} \vdash \Gamma \rhd e : \sigma_1$ *without using disjoint union types then there is some* σ_2 *such that* $\text{ML} \vdash [\Gamma]_{\cong} \rhd e : \sigma_2$ *and* $FTV(\Gamma) \vdash \sigma_1 \preceq \sigma_2$.

This theorem says that as far as typability is concerned, ML type system is as powerful as that of $\text{ML}^{\text{R}(1)}$. This can

$$(\text{VAR})\quad \Gamma\{x:\sigma\} \rhd x : \sigma$$

$$(\text{ABS})\quad \frac{\Gamma\{x:\tau_1\} \rhd e : \tau_2}{\Gamma \rhd \lambda x.e : \tau_1 \to \tau_2} \qquad (\text{APP})\quad \frac{\Gamma \rhd e_1 : \tau_1 \to \tau_2 \quad \Gamma \rhd e_2 : \tau_1}{\Gamma \rhd e_1\ e_2 : \tau_2}$$

$$(\text{PROD})\quad \frac{\Gamma \rhd e_1 : \tau_1 \quad \Gamma \rhd e_2 : \tau_2}{\Gamma \rhd (e_1, e_2) : \tau_1 \times \tau_2} \qquad (\text{PROJ}_i)\quad \frac{\Gamma \rhd e : \tau_1 \times \tau_2}{\Gamma \rhd e.i : \tau_i} \quad i \in \{1,2\}$$

$$(\text{TABS})\quad \frac{\Gamma \rhd e : \sigma}{\Gamma \rhd e : \forall T.\sigma} \text{ if } T \cap FTV(\Gamma) = \emptyset \qquad (\text{TAPP})\quad \frac{\Gamma \rhd e : \sigma_1}{\Gamma \rhd e : \sigma_2} \text{ if } \sigma_2 \preceq_{ML} \sigma_1$$

$$(\text{LET})\quad \frac{\Gamma \rhd e_1 : \sigma_1 \quad \Gamma\{x:\sigma_1\} \rhd e_2 : \tau_2}{\Gamma \rhd \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : \tau_2}$$

Figure 4: ML Type System

intuitively be understood as the property being that in a positive position a type containing a free type variables denotes the same property of the term as the corresponding closed polymorphic type. The reason why the above theorem does not hold for expressions involving disjoint unions can also be understood similarly: in a type assignment, a closed polymorphic type is strictly more general than the corresponding monotype containing free type variables.

One implication of this result is that if the only role of a type system is the static check of consistency of a program, then our type system is equivalent to that of ML (for the fragment not involving disjoint unions) and does not gain much new benefits. However, when we exploit type information at runtime, then different type systems and the different type reconstruction algorithms have different impact on the runtime semantics of the language. As we have mentioned in Introduction, the difference can be significant and sometimes exponentially large.

4 Type Inference Algorithm

For a type assignment Γ, we write ID_Γ for the identity substitution on the set $FTV(\Gamma)$ of free type variables in Γ. Figure 5 gives the type inference algorithm for $\text{ML}^{\text{R}(1)}$ as an algorithm to compute substitution S and type σ for a given e and Γ. The rationale behind this type inference algorithm is to delay type application until it is really needed. Instead of performing type application at the time of variable reference, the algorithm simply carry around a polytype. Type application is performed when it is required due to the monomorphic type restriction in rules (ABS), (APP) and (CASE). At the time of function application, for example, type instantiation is performed to the extent that the result types become conform to type shapes required by rule (APP).

On the other hand, there are some possible alternatives for type abstraction. The algorithm shown in Figure 5 abstracts the free type variables in the argument type of the function type at the time of lambda abstraction, and it abstracts all the free type variables before variable binding in let and case statement. The latter is necessary to obtain the completeness of type inference, but the former is optional. In fact, there are several possible strategies for inserting type abstraction. One extreme is to insert type abstractions of all the free type variables every time after lambda abstraction and application, and the other is to insert type abstractions only at variable bindings in let and case statements. Each different strategies yield different runtime type passing behavior, but the algorithm remains complete as far as it abstracts all the free type variables at the time of variable bindings in let and case statements. We will show more sophisticated strategy for practical programming language later.

The algorithm is sound with respect to the type system of $\text{ML}^{\text{R}(1)}$ as show in the following.

Theorem 3 *If* $(S, \sigma) = \mathcal{W}^{R(1)}(\Gamma, e)$ *then* $S(\Gamma) \rhd e : \sigma$.

The following theorem shows that the algorithm is complete with respect to the type system of $\text{ML}^{\text{R}(1)}$.

Theorem 4 *For any pair* (Γ, e)*, if there are some* S_0, σ_0 *such that* $\text{ML}^{\text{R}(1)} \vdash S_0(\Gamma) \rhd e : \sigma_0$ *then* $\mathcal{W}^{R(1)}(\Gamma, e)$ *succeeds with* (S_1, σ) *such that* $dom(S_1) = FTV(\Gamma)$*, there is some* S_2 *such that* $S_0(\Gamma) = S_2 S_1(\Gamma)$*, and* $S_0(\Gamma) \vdash \sigma_0 \preceq S_2(\sigma)$.

The extra condition $dom(S_1) = FTV(\Gamma)$ is there to make the inductive proof go through. This theorem states that the type inferred by the algorithm is more general than any other derivable types when they are considered modulo type equivalence with respect to the set of free type variables in the given type assignment.

5 Compiling $\text{ML}^{\text{R}(1)}$ to an Implementation Language

The new type inference algorithm we have developed intends to serve as a better front-end for type-directed compilation. To show the feasibility of this, we develop an algorithm to translate raw $\text{ML}^{\text{R}(1)}$ terms into an intermediate language that can implement type-passing semantics. Since type inference is not an issue in an intermediate language, we can choose a calculus with an appropriate expressive type system. In recent type-directed compilation for polymorphic functional languages including TIL [25] and FLINT [24], an explicitly typed polymorphic lambda calculus such as System F and F_ω is used as an intermediate langauge. Since we have not considered type constructors, for our purpose System F is sufficient to serve as an intermediate language for $\text{ML}^{\text{R}(1)}$.

Our implementation strategy is therefore to reconstruct an explicitly typed version of $\text{ML}^{\text{R}(1)}$ term, which we call an $\text{XML}^{\text{R}(1)}$ term, by modifying the type inference algorithm and then translate the constructed $\text{XML}^{\text{R}(1)}$ term into a

$$\mathcal{W}^{R(1)}(\Gamma, x) = \text{if } x \notin dom(\Gamma) \text{ then } failure \text{ else } (ID_\Gamma, \Gamma(x)).$$

$$\begin{array}{l}\mathcal{W}^{R(1)}(\Gamma, \lambda x.e) = \text{let } (S_1, \sigma_1) = \mathcal{W}^{R(1)}(\Gamma\{x : t\}, e_1) \ (t \text{ fresh}) \\ \qquad\qquad T = FTV(S_1(t)) \setminus FTV(S_1(\Gamma)) \\ \qquad\qquad \text{in } \ (S_1|_\Gamma, \forall T.S_1(t) \to \sigma_1).\end{array}$$

$$\begin{array}{l}\mathcal{W}^{R(1)}(\Gamma, e_1\ e_2) = \text{let } (S_1, \sigma_1) = \mathcal{W}^{R(1)}(\Gamma, e_1) \\ \qquad\qquad (S_2, \tau_1 \to \sigma_2) = \\ \qquad\qquad\quad \text{case } \sigma_1 \text{ of } \ \forall T.(\tau_0 \to \sigma_0) \Rightarrow (ID_{S_1(\Gamma)}, [T'/T](\tau_0 \to \sigma_0)) \ \ (T' \text{ fresh}) \\ \qquad\qquad\qquad\qquad\quad \mid \tau_0 \to \sigma_0 \Rightarrow (ID_{S_1(\Gamma)}, \tau_0 \to \sigma_0) \\ \qquad\qquad\qquad\qquad\quad \mid \forall t.t \Rightarrow (ID_{S_1(\Gamma)}, t_1 \to t_2) \ (t_1, t_2 \text{ fresh}) \\ \qquad\qquad\qquad\qquad\quad \mid t \Rightarrow ([t_1 \to t_2/t]ID_{S_1(\Gamma)}, t_1 \to t_2) \ (t_1, t_2 \text{ fresh}) \\ \qquad\qquad\qquad\qquad\quad \mid \text{otherwise} \Rightarrow failure \\ \qquad\qquad (S_3, \sigma_3) = \mathcal{W}^{R(1)}(S_2S_1(\Gamma), e_2) \\ \qquad\qquad \tau_2 = freshInst(\sigma_3) \\ \qquad\qquad S_4 = \mathcal{U}(\{(S_3(\tau_1), \tau_2)\}) \\ \qquad\quad \text{in } (S_4S_3S_2S_1, S_4S_3(\sigma_2)).\end{array}$$

$$\begin{array}{l}\mathcal{W}^{R(1)}(\Gamma, (e_1, e_2)) = \text{let } (S_1, \sigma_1) = \mathcal{W}^{R(1)}(\Gamma, e_1) \\ \qquad\qquad (S_2, \sigma_2) = \mathcal{W}^{R(1)}(S_1(\Gamma), e_2) \\ \qquad\quad \text{in } (S_2S_1, S_2(\sigma_1) \times \sigma_2)\end{array}$$

$$\begin{array}{l}\mathcal{W}^{R(1)}(\Gamma, e_1.i) = \text{let } (S_1, \sigma_1) = \mathcal{W}^{R(1)}(\Gamma, e_1) \\ \qquad\qquad (S_2, \sigma_2 \times \sigma_3) = \text{case } \sigma_1 \text{ of } \ \forall T.(\sigma_1^1 \times \sigma_1^2) \Rightarrow (ID_{S_1(\Gamma)}, [T'/T](\sigma_1^1 \times \sigma_1^2)) \ (T' \text{ fresh}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \sigma_1^1 \times \sigma_1^2 \Rightarrow (ID_{S_1(\Gamma)}, \sigma_1^1 \times \sigma_1^2) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \forall t.t \Rightarrow (ID_{S_1(\Gamma)}, t_1 \times t_2) \ (t_1, t_2 \text{ fresh}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mid t \Rightarrow ([t_1 \times t_2/t]ID_{S_1(\Gamma)}, t_1 \times t_2) \ (t_1, t_2 \text{ fresh}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \text{otherwise} \Rightarrow failure \\ \qquad\quad \text{in } (S_2S_1, \sigma_i)\end{array}$$

$$\mathcal{W}^{R(1)}(\Gamma, inj_1(e)) = \text{let } (S_1, \sigma_1) = \mathcal{W}^{R(1)}(\Gamma, e) \text{ in } (S_2S_1, \sigma_1 + \forall t.t)$$

$$\mathcal{W}^{R(1)}(\Gamma, inj_2(e)) = \text{let } (S_1, \sigma_1) = \mathcal{W}^{R(1)}(\Gamma, e) \text{ in } (S_2S_1, \forall t.t + \sigma_1)$$

$$\begin{array}{l}\mathcal{W}^{R(1)}(\Gamma, case\ e_0\ of\ inj_1(x_1) \Rightarrow e_1, inj_2(x_2) \Rightarrow e_2) = \\ \quad \text{let } (S, \sigma_1) = \mathcal{W}^{R(1)}(\Gamma, e_0) \\ \qquad (S_2, \sigma_2 + \sigma_3) = \text{case } \sigma_1 \text{ of } \ \forall T.(\sigma_1^1 + \sigma_1^2) \Rightarrow (ID_{S_1(\Gamma)}, [T'/T](\sigma_1^1 + \sigma_1^2)) \ (T' \text{ fresh}) \\ \qquad\qquad\qquad\qquad\quad \mid \sigma_1^1 + \sigma_1^2 \Rightarrow (ID_{S_1(\Gamma)}, \sigma_1^1 + \sigma_1^2) \\ \qquad\qquad\qquad\qquad\quad \mid \forall t.t \Rightarrow (ID_{S_1(\Gamma)}, t_1 + t_2) \ (t_1, t_2 \text{ fresh}) \\ \qquad\qquad\qquad\qquad\quad \mid t \Rightarrow ([t_1 + t_2/t]ID_{S_1(\Gamma)}, t_1 + t_2) \ (t_1, t_2 \text{ fresh}) \\ \qquad\qquad\qquad\qquad\quad \mid \text{otherwise} \Rightarrow failure \\ \qquad T_1 = FTV(S_2S_1(\Gamma)) \setminus FTV(\sigma_2) \\ \qquad (S_3, \sigma_4) = \mathcal{W}^{R(1)}(S_2S_1(\Gamma)\{x : \forall T_1.\sigma_2\}, e_1) \\ \qquad T_2 = FTV(S_3S_2S_1(\Gamma)) \setminus FTV(S_3(\sigma_3)) \\ \qquad (S_4, \sigma_5) = \mathcal{W}^{R(1)}(S_3S_2S_1(\Gamma)\{x : \forall T_2.S_3(\sigma_3)\}, e_2) \\ \qquad \tau_1 = freshInst(\sigma_4) \\ \qquad \tau_2 = freshInst(\sigma_5) \\ \qquad S_5 = \mathcal{U}(\{(S_4(\tau_1), \tau_2)\}) \\ \quad \text{in } (S_5S_4S_3S_2S_1, S_5(\tau_2)).\end{array}$$

$$\begin{array}{l}\mathcal{W}^{R(1)}(\Gamma, let\ x = e_1\ in\ e_2) = \text{let } (S_1, \sigma_1) = \mathcal{W}^{R(1)}(\Gamma, e_1) \\ \qquad\qquad\qquad\qquad T = FTV(\sigma_1) \setminus S_1(\Gamma) \\ \qquad\qquad\qquad\qquad (S_2, \sigma_2) = \mathcal{W}^{R(1)}(S_1(\Gamma)\{x : \forall T.\sigma_1\}, e_2) \\ \qquad\qquad\qquad \text{in } (S_2S_1, \sigma_2)\end{array}$$

Figure 5: Type inference algorithm.

System F term. The set of XML$^{R(1)}$terms is defined as follows.

$$\begin{array}{rcl} M & ::= & x \mid \lambda x:\tau.M \mid M\ M \mid \\ & & (M,M) \mid M.1 \mid M.2 \mid \\ & & inj_1(M:\sigma_1{+}\sigma_2) \mid inj_2(M:\sigma_1{+}\sigma_2) \mid \\ & & case\ M\ of\ inj_1(x:\sigma) \Rightarrow M, inj_2(x:\sigma) \Rightarrow M \mid \\ & & let\ x:\sigma = M\ in\ M \mid \Lambda t.e \mid M\ \mathcal{I} \end{array}$$

The type system of XML$^{R(1)}$ is obtained from that of ML$^{R(1)}$ by replacing the raw terms in each inference rule with the corresponding explicitly typed terms. As a simple extension to the result presented in [5], the type inference algorithm we presented in Section 4 can easily be modified so that it also returns an XML$^{R(1)}$ term.

The set of System F terms is given below.

$$\begin{array}{rcl} M & ::= & x \mid \lambda x:\sigma.M \mid M\ M \mid \\ & & (M,M) \mid M.1 \mid M.2 \mid \\ & & inj_1(M:\sigma_1{+}\sigma_2) \mid inj_2(M:\sigma_1{+}\sigma_2) \mid \\ & & case\ M\ of\ inj_1(x) \Rightarrow M, inj_2(x) \Rightarrow M \mid \\ & & let\ x := M\ in\ M \mid \Lambda t.e \mid M\ \sigma \end{array}$$

The major difference between ML$^{R(1)}$ and System F (besides the fact that the latter allow polymorphic functions) is that ML$^{R(1)}$ has instantiation application, which is structurally defined, while System F has type application. To compile ML$^{R(1)}$ terms to System F terms, we need to translate an instantiation application to a term containing type applications. Figure 8 gives an algorithm to perform this translation. This algorithm satisfies the following typing property.

Lemma 5 *Suppose $\Gamma \rhd e : \sigma$ is derivable in XML$^{R(1)}$ and $Inst(\sigma, \mathcal{I}) = \sigma'$ for some $\mathcal{I}$ having the same kind as that of σ then $\mathcal{IT}(\mathcal{I}, e, \sigma) = e'$ such that $\Gamma \rhd e' : \sigma'$. is a typing in System F.*

The complete translation algorithm is obtained by extending this instantiation inductively on the structure of XML$^{R(1)}$term. Its type preservation is easily verified using the above property. The resulting System F terms can then be compiled into executable code by using the techniques described, for example, in [17].

Figure 6 shows the result of type inference algorithm applied to the example given at the beginning of the paper. The first part is the actual output of our implementation, which pretty-prints the inferred type and the explicitly typed term the algorithm computed. The second part is the XML$^{R(1)}$ term corresponding to the computed explicitly typed term. As seen in this example, any unnecessary type application is not performed at the time of constructing a pair, and projection is directly applied to a pair of polymorphic functions.

6 Optimization

The reader may have noticed that the translation algorithm just described may produce some redundancy at runtime. We discuss below two major causes of redundancy together with our strategy to eliminate them.

6.1 Eliminating unnecessary type abstraction

As we have already noted, there are different strategies in inserting type abstractions are inserted. The type inference algorithm given in Figure 5 is not optimal, and sometimes unnecessary type abstraction may be inserted to some monomorphic programs. For example, for the term $(\lambda x.x)\ (\lambda x.x)$, the straightforward application of the type inference algorithm followed by the translation produces the following code:

$$((\Lambda t_1.(\lambda x:t_1.x))\ t_3 \rightarrow t_3)((\Lambda t_2.(\lambda x:t_2.x))\ t_3)$$

Apparently both of the type abstractions and type application are redundant, and should be eliminated. One simple method is to make the type inference algorithm "context sensitive" so that eager type abstraction is performed only when it will be bound to a variable. We use the standard notion of contexts (tarms with a "hole") to indicate the occurrence of expression on which the type inference algorithm is called. We [ ] for the hole of a context and write $C[e]$ for the term (or context) ontained by filling the hole of the context with e. There are the following three typical contexts we must distinguish.

- The contexts where type abstraction should perform ergaly.
  Typical contexts of this type are C[`let` x `=` [ ] `in` e] and C[`case` [ ] `of inl(`x`) =>` e_1`, inr(`y`) =>` e_2].
- The contexts where type abstraction should not perform. Typical contexts of the type are $C[e_1\ [\ ]]$, C[`case` e_1 `of inl(`x`) =>` [ ]`, inr(`y`) =>` e_2], and C[`case` e_1 `of inl(`x`) =>` e_2`, inr(`y`) =>` [ ]] where the inferred types are subject to unification.
- The contexts where type abstraction should performs *except for* a few out-most level. Typical contexts of the type include C[`let` x `=` [ ] e_1 `in` e_2] where the domain type of the inferred function type is unified and the resut type is bound to variables.

The third type of contexts depend on its sourrounding context. A careful examination of those and the type inference algorithm reveals that the appropriate contex information to control type abstraction is summarized by the number n of application performed before it is bound to a variable. To calculate this number, we consider the set of natural numbers extended with ∞ and define $\infty + n = \infty$, $\infty - n = \infty$. This context information can be computed by the simple algorithm given in Figure 7. The type inference algorithm given in Figure 5 is then modified to add the extra parameter indicating the context information n, and in inferring a type a lambda abstraction, the algorithm insert type abstraction if $n = 0$. Our prototype type inference system incorporate this context sensitive type abstraction mechanism.

6.2 Minimizing run-time data reconstruction.

Our translation of instantiation of XML$^{R(1)}$ to System F may reconstruct data structures containing polymorphic terms. For example, term $let\ x = (\lambda x.x, \lambda x.x)\ in\ ((\lambda x.x)\ \ x)$ is translated to the following System F term:

$$\begin{array}{l} \text{let} \\ \quad x : ((\forall t_1(t_1 \rightarrow t_1)) \times (\forall t_2(t_2 \rightarrow t_2))) = \\ \qquad\qquad\qquad\qquad (\Lambda t_1 \lambda(x:t_1).x, \Lambda t_2 \lambda(x:t_2).x) \end{array}$$

An output of our type inference algorithm:

```
->let f=(fn x=>x) in let x=(f,f) in let f=(#1 x) in let x=(f,f)
  in x end end end end;
>>('a.'a -> 'a) * ('a.'a -> 'a)
  let f=('a.(fn x:'a=>x))
  in let x=(f,f)
     in let f=(#1 x)
        in let x=(f,f)
           in x
           end
        end
     end
  end
```

which corresponds to the following term in $\mathrm{XML}^{\mathrm{R}(1)}$.

let f:$\forall\alpha.\alpha \to \alpha = \Lambda\alpha.\lambda$ x:α.x in
  let x:$(\forall\alpha.\alpha \to \alpha)\times(\forall\alpha.\alpha \to \alpha)$ = (f, f) in
    let f:$\forall\alpha.\alpha \to \alpha$ = x.1 in
      let x:$(\forall\alpha.\alpha \to \alpha)\times(\forall\alpha.\alpha \to \alpha)$ = (f, f) in x

Figure 6: Example of type inference

$$
\begin{array}{rcl}
P(n,[\,]) & = & n \\
P(n,\lambda x.C]) & = & P(if\ n=0\ then\ 0\ else\ n-1,C) \\
P(n,C\ e) & = & P(n+1,C) \\
P(n,e\ C) & = & P(\infty,C) \\
P(n,(C,e)) & = & P(n,C) \\
P(n,(e,C)) & = & P(n,C) \\
P(n,inj_1(C)) & = & P(n,C) \\
P(n,inj_2(c)) & = & P(n,C) \\
P(n,let\ x=C\ in\ e) & = & P(0,C) \\
P(n,let\ x=e\ in\ C) & = & P(n,C) \\
P(n,case\ C\ of\ inj_1(x)\Rightarrow e,inj_2(x)\Rightarrow e) & = & P(0,C) \\
P(n,case\ e\ of\ inj_1(x)\Rightarrow C,inj_2(x)\Rightarrow e) & = & P(\infty,C) \\
P(n,case\ e\ of\ inj_1(x)\Rightarrow e,inj_2(x)\Rightarrow C) & = & P(\infty,C)
\end{array}
$$

Figure 7: Calculating abstraction contexts to control type abstraction

in

$$((\Lambda t_3\lambda(x:t_3).x)\ \ ((t_4\to t_4)\times(t_5\to t_5))\ \ (x.1\ \ t_4, x.2\ \ t_5))$$

This is because, in our type system, polytype values are permitted to be components of products, while only monotypes are allowed to be function argument.

Although this reconstruction seems adding extra overhead in compared with ML-style polymorphism, same situation also occurs under ML-style polymorphism with runtime type-passing. In such approach, type abstractions are treated as ordinary abstractions, and evaluation of terms with type abstractions cannot proceed where they are `let`-binded. Thus, their evaluation must be done each time variables to which they are binded are mentioned. This corresponds to term reconstruction in our rank1 type system.

Furthermore, in rank 1 type system, such reconstruction happens only once before a data structure containing polymorphic values is passed to a function as a parameter. On the other hand, in ML-style type system, always variables with polytypes are mentioned, they must be instantiated to monotypes.

Thus, our type system is enough efficient in compared with ML-style polymorphism. In addition to this, instead of translating to System F, directly developping operational semantics for $\mathrm{ML}^{\mathrm{R}(1)}$will be able to achieve further optimization.

In translation to System F, instantiations are always applied strictly. This yields term reconstructions and unnecessary instantiation applications.

As an example, we consider a following term:

$$\text{let } x = (\lambda x.x, \lambda x.x) \text{ in } (\lambda x.(x.1))\ x$$

As x is a polymorphic term, x must be instantiated to a monotype before being passed to a function $(\lambda x.(x.1))$. Under strict application strategy, we need to constract a monomorphic term from x. During this reconstruction, in spite of second component of x is not used in the final result of this term, it must be applied an instantiation.

Instead of strictly applying instantiations, by introducing a special construct for values, $\mathtt{iapp}(v,\mathcal{I})$,which delays applying instantiation $\mathcal{I}$ to v, we can dalay an instantiation application until its appiled value is really necessary, and eliminate unnecessary instantiation applications.

With this special construct, we can make the evaluation of above example not to yield instantiation application for second component of x. Suppose x is evaluated to a pair of polymorphic value (v_1, v_2). Then, instead of completely being applied an instantiation $\mathcal{I}_1 \times \mathcal{I}_2$, instantiation of x is evaluated to $\mathtt{iapp}((v_1,v_2),\mathcal{I}_1\times\mathcal{I}_2)$. When projection operation is applied to this value, only one step of instantiation

for v_1 is performed, and $\texttt{iapp}(v_1, \mathcal{I}_1)$ will be returned as a result. Thus, term reconstruction concerning function value v_1 does not occur, and instantiation of v_2 is not needed also.

Note that, because of existence of value polymorphism, semantical correctness will be preserved even with this delay construct.

7 Conclusions

Under the recently emerging paradigm of type-directed compilation, polymorphic type system and the corresponding type inference algorithm may have significant impact on runtime behavior of the compiled code. We have observed that a type-directed compilation such as those implemented in the latest Standard ML New Jersey compiler sometimes produces unacceptably high runtime overhead compared to the conventional untyped implementation. We have also observed that the combination of ML polymorphism and value-only polymorphism is overly restricted. Motivated by those problem, we have extended the type system of ML with full rank 1 polymorphic types with products and disjoint unions. The type system allows data structures to contain polymorphic objects. This feature eliminates the problem of runtime overhead generated by type-passing semantics under ML polymorphism, and significantly reduces the value-only restriction. We have then developed a type inference algorithm for the extended type system and have proved its soundness and completeness. Although establishing completeness requires delicate management of type variables, the type inference itself requires surprisingly little modification to algorithm $\mathcal{W}$ as presented in [1], and therefore can be readily implemented. We have implemented the algorithm to reconstruct an explicitly typed intermediate term and have verified that it exhibits the expected behavior. We are now implementing a prototype compiler for ML with record polymorphism, which requires type-passing semantics, based on the type inference algorithm presented in this paper.

In addition to the implementation of a prototype compiler, there are a number of issues that merit further investigation. We should consider the interaction of our type inference method with other features of modern programming language such as pattern-matching. Another interesting problem is to develop a more efficient evaluation model for $\text{XML}^{\text{R}(1)}$. System F or its variants may not be completely appropriate for the intermediate language. As discussed in Section 5, translating $\text{XML}^{\text{R}(1)}$ to System F sometimes generates overhead. A better approach would be to develop a new operational semantics for type passing calculus that directly supports $\text{XML}^{\text{R}(1)}$ style type instantiation. This problem appear to have some similarity with the problem of runtime creation of structured types under the type passing semantics. One proposal to suppress the runtime overhead is to evaluate a term representing a compound type lazily [26]. We believe that a similar mechanism can be adopted to our calculus as well.

References

[1] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[2] R. Di Cosmo. Deciding type isomorphisms in a type-assignment framework. *Journal of Functional Programming*, 3(4), 485–525 1993.

[3] J.-Y. Girard. Une extension de l'interpretation de gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et théorie des types. In *Second Scandinavian Logic Symposium*. North-Holland, 1971.

[4] S. Gorn. Explicit definitions and linguistic dominoes. In J. Hart and S. Takasu, editors, *Systems and Computer Science*, pages 77–105. University of Toronto Press, 1967.

[5] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, 1993.

[6] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. ACM Symposium on Principles of Programming Languages*, 1995.

[7] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[8] T. Jim. What are principal typings and what are they good for? In *Proc. ACM Symposium on Principles of Programming Languages*, 1996.

[9] P. Kanellakis, H.G. Mairson, and J. Mitchell. Unification and ML type reconstruction. In J-L Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, 1990.

[10] A.J. Kfoury. Type reconstruction in finite rank fragments of the second-order λ-calculus. *Information and Computation*, 15(2):228–257, 1992.

[11] 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, 1993.

[12] A.J. Kfoury and J. Wells. A direct algorithm for type inference in the rank 2 fragment of the second-order lambda-calculus. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 196–207, 1994.

[13] X. Leroy. *The Objective Caml User's Manual*. INRIA Rocquencourt, B.P. 105 78153 Le Chesnay France, 1997.

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

[15] R. Milner, R. Tofte, M. Harper, and D. MacQueen. *The Definition of Standard ML*. The MIT Press, revised edition, 1997.

[16] Y. Minamide. Compilation based on a calculus for explicit type passing. In *Proceedings of Fuji International Workshop on Functional and Logic Programming*, pages 301–320, 1996.

[17] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Proc. ACM Symposium on Principles of Programming Languages*, 1998.

$$
\begin{array}{rcl}
\mathcal{IT}(*, e, \tau) & = & e \\
\mathcal{IT}(\forall S.\mathcal{I}, e, \forall t_1 \cdots t_n \tau) & = & \mathcal{IT}(\mathcal{I}, e\ \tau_1 \cdots \tau_n, S(\tau)) \quad (S = [\tau_1/t_1, \cdots, \tau_n/t_n]) \\
\mathcal{IT}(* \to \mathcal{I}, e, \tau \to \sigma) & = & (\lambda f : \tau \to \sigma . \lambda x : \tau . \mathcal{IT}(\mathcal{I}, f\ x, \sigma))\ e \\
\mathcal{IT}(\mathcal{I}_1 \times \mathcal{I}_2, e, \sigma_1 \times \sigma_2) & = & \mathit{let}\ x : \sigma_1 \times \sigma_2 = e\ \mathit{in}\ (\mathcal{IT}(\mathcal{I}_1, x.1, \sigma_1), \mathcal{IT}(\mathcal{I}_2, x.2, \sigma_2)) \\
\mathcal{IT}(\mathcal{I}_1 + \mathcal{I}_2, e, \sigma_1 + \sigma_2) & = & \mathit{case}\ e\ \mathit{of}\ \mathit{inj}_1(x) \Rightarrow \mathit{inj}_1(\mathcal{IT}(\mathcal{I}_1, x, \sigma_1)), \mathit{inj}_2(x) \Rightarrow \mathit{inj}_2(\mathcal{IT}(\mathcal{I}_2, x, \sigma_2))
\end{array}
$$

Figure 8: Translation algorithm for instantiation application

[18] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995. A preliminary summary appeared at ACM POPL, 1992 under the title "A compilation method for ML-style polymorphic record calculi".

[19] A. Ohori. Type-directed specialization of polymorphism. *Journal of Information and Computation*, 1999. To appear. A preliminary summary appeared in Proc. International Conference on Theoretical Aspects of Computer Software, Springer LNCS 1281, pages 107–137.

[20] A. Ohori and T. Takamizawa. A polymorphic unboxed calculus as an abstract machine for polymorphic languages. *Journal of Lisp and Symbolic Computation*, 10(1):61–91, 1997.

[21] J. Peterson and M. Jones. Implementing type classes. In *Proc. ACM Conference on Programming Language Design and Implementation*, 1993.

[22] J.C. Reynolds. Towards a theory of type structure. In *Paris Colloq. on Programming*, pages 408–425. Springer-Verlag, 1974.

[23] B. Saha and Z. Shao. Optimal type lifting. In *Proc. Types in Compilation, LNCS 1473*, pages 156–177, 1998.

[24] Z. Shao. Typed common intermediate format. In *USENIX Conference on Domain-Specific Languages*, 1997.

[25] D. Tarditi, G. Morrisett, P Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *ACM Conference on Programming Language Design and Implementation*, 1996.

[26] A Tolmach. Tag-free garbage collection using explicit type parameters. In *Proc. ACM Conference on Lisp and Functional Programming*, 1994.

[27] A. Wright. Simple imperative polymorphism. *Journal of Lisp and Symbolic Computation*, 8(4):343–355, 1995.