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 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. Compared with a more powerful rank 2 type inference systems based on semi-unification, the proposed type inference algorithm infers a most general type for any typable expression by using the conventional first-order unification, and it is therefore easily adopted in existing implementation of ML family of languages.

1 Introduction

ML type discipline [13, 2] 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 [4, 20], 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, 11], 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 exhibits 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 practical basis for 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 [14] and Ocaml [12], polymorphism is restricted to syntactic values (non expansive expressions). As argued in [25], 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 over restriction, excluding a number of safe and useful programs.

As a very simple example, 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 the value parts of the programs, and therefore both are safely given a polymorphic type. In the first case, the only polymorphic part is the second component of a pair, which is a value `fn x => fn y => (x + 1,ref y)` and can safely be given a polymorphic type `int -> 'a -> int * 'a ref`. The second case involves two computation steps. Since the projection `(#2 f)` simply returns the second component from a pair of values, the result type can be the same polymorphic type as the second component of the product type of `f`. We must assume that the application of the result of this projection

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

†The first author's work was partly supported by the Parallel and Distributed Processing Research Consortium, Japan.

to `1` may involve arbitrary computation. However, since both of the argument type of the function and the type of the argument are monomorphic, the result type can be the same as the range type of the function type, and thereofre the entire term can safely be given a polymorphic type `'a -> int * 'a ref`.

Unfortunately, however, both of them (and many other similar programs) are not typed as polymorphic functions 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 provides 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 implementation of type-directed specialization of polymorphism [17]. In this paradigm, type information is passed at runtime to achieve efficient implementation of polymorphic functions. This approach has been first proposed for compilation of a polymorphic record calculus [16]; several related methods have also been developed, including overloading resolution [19], intentional type analysis [7], and unboxed operational semantics [18]. Tag-free garbage collection [24] uses type information at run-time and requires a similar run-time structure.

In these approaches, type abstraction is compiled to lambda abstraction over type information and type application is compiled to lambda 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 accumulation of runtime overheads.

To understand the problem, let us consider the following simple program in a polymorphic record calculus [16].

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

The function `xval` is given a 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 a field $X : t_1$. This type abstraction is compiled to a lambda abstraction receiving the index value corresponding to the `X` field in an argument record, and an appropriate application is inserted when this type variable is instantiated. Thus the above program is compiled to the following code.

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

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

This problem is inherent in all the other approaches that use type information at run-time based on type-passing implementation of polymorphism. Our focus in this paper is not on particular polymorphic primitives, but on a general framework for efficient type-passing implementation. We concentrate on the core ML terms with products and disjoint union, and we simply assume that type abstraction and type application are compiled to lambda abstraction and lambda application to pass type information at run-time.

Figure 1 shows an example program and the corresponding explicitly typed term. In a type-passing implementation, execution of the compiled code takes the time exponential in the size of the program, while in the conventional implementation it takes linear time. In this example, the number of type variables is exponential, but this is not essential. There are terms whose typing derivation only uses linear number of type variables but their execution take exponential time in a type-passing implementation.

In an actual program, such an extreme case may not often occur. But at least, runtime overhead increases in proportion to the number of type abstractions and type applications. In a large and modular program, this may result in a significant reduction of runtime efficiency. It may be the case that for simple cases, type-passing overhead can be eliminated by some ad-hoc optimization. When code become complicated, however, we expect that such optimization is rather difficult.

1.3 Solution by rank 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 abstraction 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 terms can be constructed for the functions `f` and `g` given in the beginning 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  f) 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. In this example, the type abstraction is performed only once on a value, and therefore both terms satisfy the value polymorphism restriction.

Such an extended type system will also reduce the runtime type-passing overhead by eliminating intermediate type applications and type abstractions. For example, the following explicitly typed term can be constructed for the example program given in Figure 1.

```
let
   val x = Λt.fn x:t =>x
```

```
let
   val x = fn x => x
   val x = (x,x)
   ⋮
   val x = (x,x)
in
   (#1 (··· (#1 x)···))
end
```

Example program

```
let
   val x = $\Lambda t_1$.fn x:$t_1$ => x
   val x = $\Lambda t_1.\Lambda t_2$(x $t_1$, x $t_2$)
   ⋮
   val x = $\Lambda t_1.\ldots\Lambda t_n$(x $t_1 \cdots t_{n/2}$, x $t_{n/2} \cdots t_n$)
in
   (#1 (··· (#1 (x int $s_1$ $\cdots s_{n-1}$)···)) 1
end
```

Corresponding explicitly typed term

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

```
      val x = (x,x)
      ⋮
      val x = (x,x)
   in
      (#1 (··· (#1 x)···)) int 1
   end
```

This program performs only one type abstraction and one type instantiation, and therefore the type-passing overhead is a small constant. This is in sharp contrast with the 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 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 give a type inference algorithm $\mathcal{W}^{R(1)}$ in the style of algorithm $\mathcal{W}$ which always computes a most general type for any typable raw term, and we prove that it is sound and complete with respect to the extended type system.

3. We give optimization methods for type inference algorithm and type-passing implementation so that redundant type abstraction and instantiation are minimized.

As we discuss in details later, our type system is equivalent to that of ML for pure ML terms without using disjoint unions. This equivalence is, however, modulo a strong equivalence relation on 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 run-time behavior under type-passing semantics. Our claim is that our type inference algorithm is indeed significantly better than that of ML for type-passing semantics. Also, for a practical impure polymorphic language with value polymorphism such as Standard ML and Ocaml, our type system is strictly stronger than ML polymorphism (even without disjoint unions).

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 soundness and completeness, the type inference algorithm and its optimization requires surprisingly little modification to algorithm $\mathcal{W}$ presented in [13, 2]. 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 that of ML and is easily understandable. The proposed type inference algorithm can therefore be readily incorporated in existing implementations of type-directed compilers. 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, 11], which properly contains our type system. However, these results are either indirect [10] or involve order constraint due to semi-unification [11]. A more direct algorithm is given in [8], but it still involves order constraints on types. In a theoretical perspective, it can be argued that type inference with rank 2 polymorphism is no more complicated than that of ML. Indeed, ML type inference is intractable in the worst case [9] and the complexity of its typability is polynomial-time equivalent to a 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. It is also not at all clear how semi-unification based type inference algorithm can be incorporated in type-directed compilers currently being investigated.

As we have mentioned, our study is motivated by recent active researches on typed-directed compilation, where types (or some information of types) are passed to polymorphic functions at runtime [16, 24, 7, 23, 19, 23]. Since all those methods are based on constructing explicit type-passing calculus, they suffer from the problem of increased overhead due to repeated unnecessary type abstractions and type applications. The type inference algorithm with rank 1 polymorphism given in this paper can be used to eliminate this problem. There are recent studies on optimization of run-time type-passing mechanism [15, 21, 1]. These optimization methods aim at reducing the overhead associated with run-time type representation in the paradigm of the conventional ML type inference system, and do not address the problem we considered in the present paper. These optimization methods are complimentary to our proposal, and we believe that they can be used to optimize type-passing implementation based on our type inference algorithm.

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, and compares it with that of ML. Section 3 gives the type inference algorithm and proves its soundness and completeness. Section 4 gives an algorithm to translate raw ML terms into an implementation language using type information obtained by type inference process. Section 5 analyses other potential overheads in type-passing implementation which is revealed by our analysis, and gives optimization methods to minimize them.

2 The $\mathrm{ML}^{\mathrm{R}(1)}$ type system

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

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.e \mid e\ e \mid \\ & & (e,e) \mid e.1 \mid e.2 \mid \mathit{inj}_1(e) \mid \mathit{inj}_2(e) \mid \\ & & \mathit{case}\ e\ \mathit{of}\ \mathit{inj}_1(x) \Rightarrow e, \mathit{inj}_2(x) \Rightarrow e \mid \\ & & \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \end{array}$$

(e_1, e_2) is a pair, $e.1, e.2$ are the first and the second projection, respectively. $\mathit{inj}_1(e), \mathit{inj}_2(e)$ 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 [2] 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 types 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 $t \notin \mathit{dom}(S)$, 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 substitutions S_1, S_2, written as S_2S_1, is defined as a substitution S' such that $\mathit{dom}(S') = \mathit{dom}(S_1)$, and for any $t \in \mathit{dom}(S_1)$, $S_2(S_1(t))$ where S_2 is regarded as the unique extension of S_2. We assume that composition associates 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 type variables of polytypes: (1) all bound type variables are distinct and different from any free type variables, and (2) any polytype of the form $\forall T.\sigma$ satisfies $T \subseteq FTV(\sigma)$. The first condition is the usual "bound variable convention" for bound type variables.

In order to define a type system of $\mathrm{ML}^{\mathrm{R}(1)}$ we need to define a polymorphic instantiation relation. Instantiation involves substitution for bound type variables. Because of this, we need to work with 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 canonical representation of its equivalence class. It is easy to come up with a representation. Below, we outline the one 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. This guarantees that bound type variables are always distinct from free type variables. For $i \leq j$, we write (i, j) for the set of natural numbers from i through j. Let $\forall T.\sigma$ be a given polytype satisfying the bound variable convention. Let $\overline{\sigma}$ be a canonical representation of σ. We define a linear order on T 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 [5] 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 $\overline{\sigma}$. A representation $\overline{\forall T.\sigma}$ of $\forall T.\sigma$ is then given as follows.

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

The following is an example.

$$\overline{\forall\{s,t\}.t \to (\forall\{t,u\}.t \to s{\times}u)}$$

$$= \quad \forall(3,4).3 \to (\forall(1,2).1 \to 4{\times}2)$$

It is easy to give a recursive algorithm to compute a canonical representation.

We turn to the definition of instantiation under the assumption that polytypes are their canonical representations. 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:

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

where $*$ is the empty instantiation and $S.\mathcal{I}$ is one that performs instantiation for a polytype of the form $\forall T.\sigma$ such that $dom(S) = T$. To define applicability of instantiations 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.

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

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

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

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

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

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

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

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

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

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.

$$\begin{array}{l} Inst(\forall\{t_1\}.t_1 \to (\forall\{t_2\}.t_2 \to t_1), [int/t_1].* \to [bool/t_2].*) \\ \quad = \quad int \to (bool \to int) \end{array}$$

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

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

Using these definitions and notations, the type system of $\mathrm{ML}^{\mathrm{R}(1)}$ is given in Figure 2 as a proof system to derive a *typing judgment* of the form $\Gamma \rhd e : \sigma$ indicating the fact that e has type σ under type assignment 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 is 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 before they are bound 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, and consider type inference for a case statement *case* e_1 *of* $inj_1(x) \Rightarrow e_2, inj_2(y) \Rightarrow e_3$. For a type inference algorithm to be sound, the algorithm must assume for x and y 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 ordering $\le$. In a unification based type inference system, however, components of an inferred type are usually not maximal with respect to the ordering $\le$. Note that the rule (TABS) is not sufficient to achieve maxmality in σ_1 and σ_2 separately, since this rule only generalizes 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 [3].

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. A free type variable context lists those type variables that may appear free in other types such as those in a 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. We write

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

to denote the fact that σ_1 is equivalent to σ_2 with respect to a free type variables context C. This relation is defined in two stages. We first define a relation $\sigma_1 \cong \sigma_2$. Let F range over contexts of type expressions 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 $\sigma \cong \sigma'$ is the reflexive, symmetric and transitive closure of the following axiom scheme

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

where we assume bound type variable convention. Any $\mathrm{ML}^{\mathrm{R}(1)}$ polytype is equivalent to a unique ML polytype

$$(\textsc{var})\quad \Gamma\{x:\sigma\} \rhd x : \sigma \qquad\qquad (\textsc{abs})\quad \frac{\Gamma\{x:\tau\} \rhd e : \sigma}{\Gamma \rhd \lambda x.e : \tau \to \sigma}$$

$$(\textsc{app})\quad \frac{\Gamma \rhd e_1 : \tau \to \sigma \quad \Gamma \rhd e_2 : \tau}{\Gamma \rhd e_1\ e_2 : \sigma} \qquad\qquad (\textsc{prod})\quad \frac{\Gamma \rhd e_1 : \sigma_1 \quad \Gamma \rhd e_2 : \sigma_2}{\Gamma \rhd (e_1, e_2) : \sigma_1 \times \sigma_2}$$

$$(\textsc{proj}_i)\quad \frac{\Gamma \rhd e : \sigma_1 \times \sigma_2}{\Gamma \rhd e.i : \sigma_i} \quad i \in \{1,2\} \qquad\qquad (\textsc{inj}_i)\quad \frac{\Gamma \rhd e : \sigma_i}{\Gamma \rhd \mathit{inj}_i(e) : \sigma_1 + \sigma_2} \quad i \in \{1,2\}$$

$$(\textsc{case})\quad \frac{\Gamma \rhd e_1 : \sigma_1 + \sigma_2 \quad \Gamma\{x : \forall T_1.\sigma_1\} \rhd e_2 : \tau \quad \Gamma\{y : \forall T_2.\sigma_2\} \rhd e_3 : \tau}{\Gamma \rhd \mathit{case}\ e_1\ \mathit{of}\ \mathit{inj}_1(x) \Rightarrow e_2, \mathit{inj}_2(y) \Rightarrow e_3 : \tau} \quad (T_1 \cup T_2) \cap FTV(\Gamma) = \emptyset$$

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

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

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

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

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

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

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

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 (up to α-equivalence) σ' satisfying the following: $C \vdash \sigma \cong \sigma'$, σ' is an ML polytype 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 relations on polytypes as follows.

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

For the convenience of the following development, we use the notation $\mathit{freshInst}(\sigma)$ 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 $C \vdash \sigma \cong \mathit{freshInst}(\sigma)$.

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 $\mathrm{ML}^{\mathrm{R}(1)}$ 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[2] type system, denoted here by $\preceq_{ML}$, is characterized as

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

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, $\mathrm{ML}^{\mathrm{R}(1)}$ type system is strictly more powerful. This is because, in $\mathrm{ML}^{\mathrm{R}(1)}$, variable binding in case branches is polymorphic.

Damas-Milner type system for ML is given in Figure 3. Since $\mathrm{ML}^{\mathrm{R}(1)}$ type system is an extension of that of ML, it is immediate that any ML derivation is also a derivation of $\mathrm{ML}^{\mathrm{R}(1)}$. We can show that for expressions not involving disjoint union types, the converse also holds under our interpretation of type equivalence and polytype ordering. This is analogous to the result shown in [10] that the rank 2 system Λ_2 is equivalent to a restricted rank 2 system Λ_2^-. 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* $\mathrm{ML}^{\mathrm{R}(1)} \vdash \Gamma \rhd e : \sigma_1$ *without using disjoint union types then there is some* σ_2 *such that* $\textsc{ml} \vdash [\Gamma]_{\cong} \rhd e : \sigma_2$ *and* $FTV(\Gamma) \vdash \sigma_1 \preceq \sigma_2$.

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 ML terms not involving disjoint unions and impure fetures) 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. Also, one should note that this equivalence does not even hold under value polymorphism restriction since enlarging the scope of

$$
\begin{array}{llll}
(\textsc{var}) & \Gamma\{x:\sigma\} \rhd x : \sigma & (\textsc{abs}) & \dfrac{\Gamma\{x:\tau_1\} \rhd e : \tau_2}{\Gamma \rhd \lambda x.e : \tau_1 \to \tau_2} \\[2ex]
(\textsc{app}) & \dfrac{\Gamma \rhd e_1 : \tau_1 \to \tau_2 \quad \Gamma \rhd e_2 : \tau_1}{\Gamma \rhd e_1\ e_2 : \tau_2} & (\textsc{prod}) & \dfrac{\Gamma \rhd e_1 : \tau_1 \quad \Gamma \rhd e_2 : \tau_2}{\Gamma \rhd (e_1, e_2) : \tau_1 \times \tau_2} \\[2ex]
(\textsc{proj}_i) & \dfrac{\Gamma \rhd e : \tau_1 \times \tau_2}{\Gamma \rhd e.i : \tau_i} \quad i \in \{1,2\} & (\textsc{tabs}) & \dfrac{\Gamma \rhd e : \sigma}{\Gamma \rhd e : \forall T.\sigma} \quad \text{if } T \cap FTV(\Gamma) = \emptyset \\[2ex]
(\textsc{tapp}) & \dfrac{\Gamma \rhd e : \sigma_1}{\Gamma \rhd e : \sigma_2} \quad \text{if } \sigma_2 \preceq_{ML} \sigma_1 & (\textsc{let}) & \dfrac{\Gamma \rhd e_1 : \sigma_1 \quad \Gamma\{x:\sigma_1\} \rhd e_2 : \tau_2}{\Gamma \rhd \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : \tau_2}
\end{array}
$$

Figure 3: ML Type System

type quantification ($\forall T$) may violate value restriction. For example, under value polymorphism restriction, the term

$$\mathit{let}\ f = (\lambda x.\lambda y.y)\ 1\ \mathit{in}\ (f\ 1, f\ \text{"1"})$$

is typable in $\text{ML}^{\text{R}(1)}$ but not in ML and is rejected by existing ML compilers.

3 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. We let $\mathcal{U}$ be the standard unification algorithm on free algebra of monotypes such that $\mathcal{U}(\tau_1, \tau_2)$ returns a most general unifier for τ, τ' if they are unifiable otherwise it return *failure.* Figure 4 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 carries 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).

For inserting type abstractions, there are possible alternatives. The algorithm shown in Figure 4 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. One extreme alternative is to insert type abstractions of all the free type variables every time after lambda abstraction and application. The other is to insert type abstractions only at variable bindings in let and case statements. Each different strategy yields different runtime type-passing behavior, but the algorithm remains sound and complete as far as it abstracts all the free type variables at the time of variable bindings in let and case statements. In Section 5, we will give an optimized type inference algorithm with more sophisticated strategy for controlling type abstraction for practical programming language implementation.

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

4 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.

In recent type-directed compilation for polymorphic functional languages including TIL [23] and FLINT [22], this process is done by constructing an explicitly typed term using the type information obtained by type inference, and then converting the explicitly typed term into a variant of explicitly-typed polymorphic lambda calculus such as System F or F_ω.

The set of explicitly typed terms, which we call $\text{XML}^{\text{R}(1)}$ terms, corresponding to the set of $\text{ML}^{\text{R}(1)}$ terms, is defined as follows.

$$
\begin{array}{lcl}
M & ::= & x \mid \lambda x:\tau.M \mid M\ M \mid \\
& & (M, M) \mid M.1 \mid M.2 \mid \\
& & \mathit{inj}_1(M : \sigma_1 + \sigma_2) \mid \mathit{inj}_2(M : \sigma_1 + \sigma_2) \mid \\
& & \mathit{case}\ M\ \mathit{of}\ \mathit{inj}_1(x) \Rightarrow M, \mathit{inj}_2(x) \Rightarrow M \mid \\
& & \mathit{let}\ x = M\ \mathit{in}\ M \mid \Lambda\{t_1, \ldots, t_n\}.M \mid M\ \mathcal{I}
\end{array}
$$

The type system of $\text{XML}^{\text{R}(1)}$ is obtained from that of $\text{ML}^{\text{R}(1)}$ by replacing the raw terms in each inference rule with the corresponding explicitly typed terms. As a simple extension to the result presented in [6], the type inference algorithm we presented in Section 3 can easily be modified so that it also returns an $\text{XML}^{\text{R}(1)}$ term.

In this section, instead of defining an operational semantics for $\text{XML}^{\text{R}(1)}$ directly, we give a translation algorithm

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

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

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

$\mathcal{W}^{R(1)}(\Gamma, (e_1, e_2)) =$ let $(S_1, \sigma_1) = \mathcal{W}^{R(1)}(\Gamma, e_1)$
                              $(S_2, \sigma_2) = \mathcal{W}^{R(1)}(S_1(\Gamma), e_2)$
                          in $(S_2S_1, S_2(\sigma_1) \times \sigma_2)$

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

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

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

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

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

In the above, $\mathcal{U}$ is the standard unification algorithm for free algebra of monotypes.

Figure 4: Type inference algorithm

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

Figure 5: Translation algorithm for instantiation application

from $\mathrm{XML}^{\mathrm{R}(1)}$ to System F. As we shall discuss in Section 5, this translation reveals another source of optimization, which our rank 1 type inference system offers.

The set of System F terms is given below.

$$
\begin{array}{rcl}
M & ::= & x \mid \lambda x : \sigma.M \mid M\ M \mid \\
 & & (M, M) \mid M.1 \mid M.2 \mid \\
 & & \mathit{inj}_1(M : \sigma_1 + \sigma_2) \mid \mathit{inj}_2(M : \sigma_1 + \sigma_2) \mid \\
 & & \mathit{case}\ M\ \mathit{of}\ \mathit{inj}_1(x) \Rightarrow M, \mathit{inj}_2(x) \Rightarrow M \mid \\
 & & \mathit{let}\ x = M\ \mathit{in}\ M \mid \Lambda\{t_1, \ldots, t_n\}.e \mid M\ \sigma
\end{array}
$$

This is a minor variant of System F with multiple type variable abstraction $\Lambda\{t_1, \ldots, t_n\}.M$ and a term of the form $\mathit{let}\ x = M_1\ \mathit{in}\ M_2$, which is considered as a shorthand for $(\lambda x : \sigma.M_2)\ M_1$ where σ is the type of M_2 uniquely determined. The type system of the above set of terms is standard.

The major difference between $\mathrm{ML}^{\mathrm{R}(1)}$ and System F (besides the fact that the latter allows polymorphic functions of higher rank) is that $\mathrm{ML}^{\mathrm{R}(1)}$ has instantiation application, which is structurally defined, while System F has type application. To compile an $\mathrm{ML}^{\mathrm{R}(1)}$ term M to the corresponding System F term $(M)^o$, we need to translate an instantiation application to a term containing type applications. Figure 5 gives an algorithm to perform this translation. For a System F term $(M)^o$ corresponding to a term M of type σ, $\mathcal{IT}(\mathcal{I}, (M)^o, \sigma)$ returns the System F term corresponding to $M\ \mathcal{I}$. The extra parameter σ, which is computed uniquely from M, is needed to make correct type annotation. This algorithm satisfies the following typing property.

Lemma 5 *Suppose $\Gamma \rhd M : \sigma$ is derivable in System F. For any $\mathcal{I}$ having the same kind as that of σ, if $\sigma' = \mathit{Inst}(\sigma, \mathcal{I})$ and $M' = \mathcal{IT}(\mathcal{I}, M, \sigma)$ then $\Gamma \rhd M' : \sigma'$ is derivable in System F.*

The algorithm to translate $\mathrm{XML}^{\mathrm{R}(1)}$ term to the corresponding System F term is obtained by extending the following phrase according to the structure of $\mathrm{XML}^{\mathrm{R}(1)}$ term M.

$$(M\ \mathcal{I})^o = \mathcal{IT}(\mathcal{I}, (M)^o, \sigma)$$

Its type preservation is easily verified using the above property.

Figure 6 shows the result of type inference algorithm applied to the example given in Figure 1 in Introduction. 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.

The source program:

$$
\begin{array}{l}
\mathit{let}\ \ x = \lambda x.x\ \ \mathit{in} \\
\mathit{let}\ \ x = (x, x)\ \ \mathit{in} \\
\quad \vdots \\
\mathit{let}\ \ x = (x, x)\ \ \mathit{in} \\
\quad (\cdots(x.1).1\cdots).1)\ \ 1
\end{array}
$$

The reconstructed explicitly typed term:

$$
\begin{array}{l}
\mathit{let}\ \ x = \Lambda\{t\}\lambda x : t.x\ \ \mathit{in} \\
\mathit{let}\ \ x = (x, x)\ \ \mathit{in} \\
\quad \vdots \\
\mathit{let}\ \ x = (x, x)\ \ \mathit{in} \\
\quad (\cdots(x.1).1\cdots).1)\ \ \mathit{int}\ \ 1
\end{array}
$$

Figure 6: Example of type inference

5 Optimizations

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.

5.1 Eliminating unnecessary type abstraction

As we have already noted, there are different strategies in inserting type abstractions. The type inference algorithm given in Figure 4 is not optimal in this respect, and sometimes unnecessary type abstractions and type applications 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:

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

Apparently both of the type abstractions and type application are redundant, and should be eliminated.

One simple but effective method is to make the type inference algorithm "context sensitive" so that eager type abstraction is performed only when the type inference algorithm is called from a context where the inferred type will be bound to a variable. We use the standard notion of contexts (terms with a "hole") to indicate *type inference contexts*, i.e. those where the type inference algorithm is called. We write $[\ ]$ for the hole of a context and write $C[e]$ $(C[C'])$ for the term (the context) obtained by filling the hole of the context with e (C').

To eliminate redundant type abstractions, we must distinguish the following three sorts of type inference contexts.

$$\begin{array}{rcl}
P(n,[\,]) & = & n\\
P(n,\lambda x.C) & = & P(\mathit{if}\ n=0\ \mathit{then}\ 0\ \mathit{else}\ n-1,C)\\
P(n,C\ e) & = & P(n+1,C)\\
P(n,e\ C) & = & P(\infty,C)\\
P(n,(C,e)) & = & P(n,C)\\
P(n,(e,C)) & = & P(n,C)\\
P(n,\mathit{inj}_1(C)) & = & P(n,C)
\end{array}
\qquad
\begin{array}{rcl}
P(n,\mathit{inj}_2(c)) & = & P(n,C)\\
P(n,\mathit{let}\ x=C\ \mathit{in}\ e) & = & P(0,C)\\
P(n,\mathit{let}\ x=e\ \mathit{in}\ C) & = & P(n,C)\\
P(n,\mathit{case}\ C\ \mathit{of}\ \mathit{inj}_1(x)\Rightarrow e,\mathit{inj}_2(x)\Rightarrow e) & = & P(0,C)\\
P(n,\mathit{case}\ e\ \mathit{of}\ \mathit{inj}_1(x)\Rightarrow C,\mathit{inj}_2(x)\Rightarrow e) & = & P(\infty,C)\\
P(n,\mathit{case}\ e\ \mathit{of}\ \mathit{inj}_1(x)\Rightarrow e,\mathit{inj}_2(x)\Rightarrow C) & = & P(\infty,C)
\end{array}$$

Figure 7: Calculating summary information of a type inference context

1. The first is those where the inferred type will be bound to a variable and therefore type abstraction should be performed eagerly. This sort of contexts include C[`let` x `=` [ ] `in` e] and C[`case` [ ] `of inl(`x`)` `=>` e_1`, inr(`y`)` `=>` e_2].

2. The second is those where the inferred types are subject to unification and therefore type abstraction should not be performed. This sort of contexts include $C[e_1\ [\ ]]$, C[`case` e_1 `of inl(`x`) =>` [ ]`, inr(`y`)` `=>` e_2], and C[`case` e_1 `of inl(`x`) =>` e_2`, inr(`y`) =>` [ ]].

3. The third is those contexts where unification is performed on some components of the inferred types. Typical contexts of this sort are those of the form $C[[\ ]\ e]$ where $C[\ ]$ is any context of type 1 or type 3.

A careful examination of those reveals that the necessary information on type inference contexts to control type abstraction is summarized by a natural number n of applications performed before the term 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 4 is then modified to add the extra parameter indicating the context information n, and in inferring a type of a lambda abstraction, the algorithm inserts type abstraction only if $n = 0$.

Our prototype type inference system incorporates this context sensitive type abstraction mechanism. Figure 8 shows an interactive session in our prototype implementation for type inference and reconstruction of explicitly typed term. The imput to the system is prompted by `->` and the output is preceded by `>>`. In the output, a term of the form `('a.`e`)` is type abstraction, `inst` e `with` $\mathcal{I}$ is type instantiation (where the empty instantiation $*$ is omitted), and a type of the form `('a,'b,....`τ`)` is a polymorphic type. The first two are the example programs discussed in Introduction. As seen from these examples, our type inference algorithm together with the context sensitive optimization produces desired explicitly typed terms representing rank 1 polymorphism with little type-passing overhead. The third example shows how rank 1 polymorphism interacts with disjoint union. `'D` appearing in this example is a free type variable which does not appear in the final result and therefore is not generalized.

5.2 Minimizing run-time data reconstruction

Another source of optimization is run-time reconstruction of data structures at the time of type instantiation. To see the problem, consider the following raw ML term. $\mathit{let}\ x = (\lambda x.x, \lambda x.x)\ \mathit{in}\ (\lambda x.(x.1\ 1))\ x$, for which the following explicitly typed term is reconstructed by the type inference algorithm.

$$\begin{array}{l}
\mathit{let}\ x = (\Lambda t_1 \lambda x : t_1.x, \Lambda t_2 \lambda x : t_2.x)\\
\mathit{in}\ (\lambda x : (\mathit{int} \to \mathit{int}) \times (t_3 \to t_3).(x.1\ 1))\\
\quad (x\ [\mathit{int}/t_1].* \times [t_3/t_2].*)\\
\mathit{end}
\end{array}$$

If we straightforwardly apply the translation algorithm, it results in the following System F term

$$\begin{array}{l}
\mathit{let}\ x = (\Lambda t_1 \lambda x : t_1.x, \Lambda t_2 \lambda x : t_2.x)\\
\mathit{in}\ (\lambda x : (\mathit{int} \to \mathit{int}) \times (t_3 \to t_3).(x.1\ 1))\ (x.1\ \mathit{int}, x.2\ t_3)\\
\mathit{end}
\end{array}$$

where a pair $(x.1\ t_3, x.2\ t_4)$ is reconstructed at run-time.

We note that this overhead is *not* specific to our rank 1 polymorphic type system. Since a polymorphic object is compiled to lambda abstraction under a type-passing semantics, in ML type system, x is compiled to a function that constructs a pair at run-time and therefore a pair is generated every time x is instantiated. The situation is generally better in our rank 1 type system, where type instantiation occurs less often. Nonetheless, we would like to minimize this run-time overhead, which does not exist in the conventional untyped operational semantics.

Our strategy is to develop a more efficient operational semantics for $\mathrm{ML}^{\mathrm{R}(1)}$ directly, instead of translating it to System F. One promising strategy for such an efficient operational semantics is to *delay* instantiation until really needed. This is inspired by a type-passing implementation method [24] where type application is evaluated lazily. In the above example of $\mathrm{ML}^{\mathrm{R}(1)}$ term, instead of evaluating the instantiation $(x\ [\mathit{int}/t_1].* \times [t_3/t_2].*)$, we can suspend this computation. Since the denotation of this instantiation is a pair of instantiated function, operation performed on this term is projection. We also know that the denotation of x is also a pair of polymorphic functions. We can then compile projection on this value to the operation that first performs projection on x followed by instantiation.

This observation yields the following evaluation strategy. Suppose x is evaluated to a pair of polymorphic value (v_1, v_2). Instead of performing instantiation $(x\ \mathcal{I}_1 \times \mathcal{I}_2)$ eagerly, we represent instantiation of the form $\mathcal{I}_1 \times \mathcal{I}_2$ as a pair of instantiations $(\mathcal{I}_1, \mathcal{I}_2)$, and we treat the pair $((v_1, v_2), (\mathcal{I}_1, \mathcal{I}_2))$ as a value of "suspended instantiation",

```
->let x = fn x => x in
  let x = (x,x) in
  let x = (x,x) in
      ((#1 (#1 x)) 1)
  end end end;
>>let x=('a.fn x:'a=>x)
  in let x=(x,x)
     in let x=(x,x)
        in (inst (#1 (#1 x)) with ([int/'a]) 1)
        end
     end
  end
  : int

->let f = ((fn x => x) 1,
           fn x =>fn y => (x + 1,y))
  in (#2 f) end;
>>let f=((fn x:int=>x) 1,
         fn x:int=>('a.fn y:'a=>(x + 1,y)))
  in (#2 f)
  end
  : int -> ('a.'a -> int * 'a)

->let a = inr(fn x => x) in
      (case a of inl(x) =>(0,"0"),
                 inr(x) => (x 1, x "1"))
  end;
>>let a=('b.Inr('a.fn x:'a=>x):'b + ('a.'a -> 'a))
  in (case inst a with (['D/'b]) of
        inl(x)=>(0,"0")
      | inr(x)=>((inst x with ([int/'a]) 1),
                 (inst x with ([string/'a]) "1")))
  end
  : int * string
```

Figure 8: Example of type reconstruction with type abstraction optimization

and delay actual instantiation until projection. Projection on such a suspended instantiation is done by first projecting both the pair of polymorphic values and the pair of instantiations on the desired component, and then performing actual instantiation. Since instantiation does not involve any user-level computation, delaying instantiation does not change the order of evaluation. For example, this strategy performs the following evaluation

$$\begin{aligned}
&(x\ [int/t_1].* \times [t_3/t_2].*).1 \\
&\overset{*}{\Longrightarrow}\ ((v_1, v_2), ([int/t_1].*, [t_3/t_2].*)).1 \\
&\Longrightarrow\ (v_1, [int/t_1].*) \\
&\Longrightarrow\ [int/t_1]v_1
\end{aligned}$$

under a run-time value environment that maps x to (v_1, v_2). This strategy can be adopted to other data structures as well.

We hope that with this optimization, rank 1 polymorphism can eliminate most of the run-time overhead of type-passing implementation of polymorphism. Formal definition of the operational semantics and a detailed implementation are under way.

6 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 realizing type-passing operational semantics 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 problems, 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 polymorphism 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 algorithm itself requires surprisingly little modification to algorithm $\mathcal{W}$ as presented in [13, 2], and therefore can be readily implemented. We have also presented optimization methods which further reduce type-passing overhead and suppress redundant run-time data reconstruction. With these optimizations, we believe that the type inference system presented in this paper can serve as a basis for efficient implementation of polymorphic languages.

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. We plan to report a fuller description of the method described in this paper together with the result of implementation elsewhere.

References

[1] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *Proc. International Conference on Functional Programming*, 1998.

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

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

[4] 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.

[5] 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.

[6] 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.

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

[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 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.

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

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

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

[15] 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.

[16] 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".

[17] 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.

[18] 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.

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

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

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

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

[23] 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.

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

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

A Direct Algorithm for Type Inference in the Rank-2 Fragment of the Second-Order λ-Calculus[*]

A. J. Kfoury
kfoury@cs.bu.edu
Dept. of Computer Science
Boston University

J. B. Wells
jbw@cs.bu.edu
Dept. of Computer Science
Boston University

Abstract

We examine the problem of type inference for a family of polymorphic type systems containing the power of Core-**ML**. This family comprises the levels of the stratification of the second-order λ-calculus (system **F**) by "rank" of types. We show that typability is an undecidable problem at every rank $k \geq 3$. While it was already known that typability is decidable at rank 2, no direct and easy-to-implement algorithm was available. We develop a new notion of λ-term reduction and use it to prove that the problem of typability at rank 2 is reducible to the problem of acyclic semi-unification. We also describe a simple procedure for solving acyclic semi-unification. Issues related to principal types are discussed.

1 Introduction

Background and Motivation. Many modern functional programming languages use polymorphic type systems that support automatic type inference. Automatic type inference for untyped or partially typed programs saves the programmer the work of specifying the type of every identifier. Type polymorphism lets the programmer write generic functions that work uniformly on arguments of different types and it thus avoids the maintenance problem that results from duplicating similar program code at different types. The first programming language to use polymorphic type inference was the functional language **ML** [GMW79, Mil85]. Due to its usefulness, many of the aspects of **ML** have been subsequently incorporated in other languages (e.g. Miranda [Tur85], Haskell [HW88]). **ML** shares with Algol 68 properties of compile-time type checking, strong typing and higher-order functions while also providing automatic type inference and type polymorphism.

The usefulness of a particular polymorphic type system depends very much on how feasible the task of type inference is. We define concepts in terms of the λ-calculus, which we use as our pure functional programming language throughout this paper. By *type inference* we mean the problem of finding a type derivable for a λ-term in the type system. The problem of type inference involves several issues:

(1) Is *typability* decidable, i.e. is it decidable whether any type at all is derivable for a λ-term in the type system?

If typability is undecidable, then there is little more to say in relation to type inference. (Although a programming language may work around this problem by asking the programmer to supply part of the type information and by using heuristics, we will omit discussion of this possibility.) Otherwise, if typability is decidable, then it is possible to construct a type for typable λ-terms, i.e. type inference can be performed, in which case we further ask:

(2) How efficiently can deciding typability and performing type inference be done?

The answer to this question determines whether the type system is feasible to implement. Another related issue is:

(3) Can a *principal type* (a "most general" type) be constructed for typable λ-terms?

Closely related to the issue of principal types is *type checking*, the problem of deciding, given a λ-term M and a type τ, whether τ is one of the types that may be derived for M by the type system under consideration.

In addition to the feasibility of a particular polymorphic type system, its usefulness also depends on how much flexibility the type system gives the programmer. Although the polymorphism of **ML** is useful, it is too weak to assign types to some program phrases that are natural for programmers to write. To overcome these limitations researchers have investigated the feasibility of type systems whose typing power is a superset of that of **ML**. Over the years, this line of research has dealt with various polymorphic type systems for functional languages and λ-calculi, in particular the powerful type system of the Girard/Reynolds second-order λ-calculus [Gir72, Rey74], which we will call by its other name, System **F**. In the long quest to settle the type checking and typability problems for **F**, researchers have also considered the problem for **F** modified by various restrictions. Multiple stratifications of **F** have been proposed, e.g. by depth of bound type variable from binding quantifier [GRDR91] and by limiting the number of generations of instantiation of quantifiers themselves introduced by instantiation [Lei91]. One natural restriction which we consider in this paper results from stratifying **F** according to the "rank" of types

*This work is partly supported by NSF grant CCR–9113196.

To appear in the Proceedings of the 1994 ACM Conference on LISP and Functional Programming, June 27–29, 1994, Orlando, Florida, USA.

allowed in the typing of λ-terms and restricting the rank to various finite values (introduced in [Lei83] and further studied in [McC84, KT92]). All of these systems improve on the expressive power of **ML**.

Unfortunately, it is often the case that the more flexible and powerful a particular polymorphic type system is, the more likely that automatic type inference will be infeasible or impossible. As discouraging examples, the problems of typability and type checking for many of the polymorphic type systems mentioned above have recently been proven undecidable. Type checking and typability were shown to be undecidable for System **F** (cf. recent results submitted for publication elsewhere [Wel93]) and for its very powerful extension, System $\mathbf{F}_\omega$ [Urz93]. Other related systems that are not exactly extensions of **ML** have also recently been proven to have undecidable typability, i.e. System $\mathbf{F}_\leq$ which relates to object-oriented languages [Pie92], and the $\lambda\Pi$-calculus which relates to extensions of λ-Prolog [Dow93].

Against this recent background, it is desirable to demarcate precisely where the boundary between decidable and undecidable typability lies within various stratifications of System **F**. In the case of decidable typability, it is also desirable to develop simple and easy-to-implement algorithms for the most powerful level within a stratification that is also feasible to use. We undertake this task for the stratification of System **F** by rank of types.

Contributions of This Paper. We can now firmly establish the boundary for decidability of typability and type checking within the stratification of System **F** by rank of types. The two problems are undecidable for every fragment of **F** of rank ≥ 3 and are decidable for rank ≤ 2. The undecidability of type checking at rank ≥ 3 can be seen by observing that the proof for the undecidability of type checking in **F** in [Wel93] requires only rank-3 types. The undecidability of typability at rank ≥ 3 results from the fact that the constants c and f defined in section 5 of [KT92] can be encoded using the methods of [Wel93] in System Λ_3 (the rank-3 fragment of **F**) and from Theorem 30 of [KT92]. We give this encoding in this paper. Since it was already known from [KT92] that typability is decidable for System Λ_2 (the rank-2 fragment of **F**), we know exactly where the boundary of decidability for typability lies. The Systems Λ_1 and Λ_0 are both equivalent to the simply-typed λ-calculus and are uninteresting. These circumstances lead us to look for a simple and direct algorithm for type inference within Λ_2.

The existing proof that typability is decidable for System Λ_2 uses a succession of several reductions to the typability problem in **ML** and results in a type inference algorithm that is neither simple nor easy to understand. Since this previous algorithm is a reduction to the type inference algorithm of **ML**, it can not possibly be as simple. In this paper, we give a simpler and more direct algorithm for the decidable case of typability in Λ_2 which does not depend on the type inference algorithm of **ML**. We first prove that Λ_2 is equivalent to a restriction named System $\Lambda_2^{-,*}$ having many convenient properties. We then develop a notion of reduction named θ which converts λ-terms into a form which is more amenable to type inference but which also preserves every λ-term's set of derivable types in $\Lambda_2^{-,*}$. We define a further restricted System $\Lambda_2^{-,*,\theta}$ to take advantage of this. The type inference problem in $\Lambda_2^{-,*,\theta}$ for a λ-term in θ-normal form is easily converted into an instance of the acyclic semi-unification problem. Finally, we give a simple algorithm for solving the acyclic semi-unification problem. The complexity of the whole procedure is the same as that of type inference in **ML**.

The principal typing situation for Λ_2 is not as nice as for **ML**. For a given λ-term, there is no principal type such that every type derivable for the λ-term can be seen as a substitution instance of the type. We show there is a weak form of principal typing where the free type variables of a type can have open types substituted for them, but this does not allow a single type to generate all of the possible types for a λ-term. Quirks of the typing system that occur due to the lack of principal types are discussed.

We omit proofs of lemmas and theorems in this conference report to remain within the page limit. We postpone to either a later extended version of this paper or another paper discussion of the relationship between **ML**-typability and typability in Λ_2 and of type checking in Λ_2.

Acknowledgements. A number of definitions used in this paper were lifted from [KT92, KTU90, KTU93], so credit goes to both Tiuryn and Urzyczyn.

2 System Λ_k and System Λ_2^-

In this section, we define first the untyped λ-calculus, then System **F**, then the restriction of System **F** that results in System Λ_k. Then, we define a restriction of System Λ_2 called System Λ_2^- which has equivalent typing power.

In our presentation, we use the "Curry view" of type systems for the λ-calculus, in which pure terms of the λ-calculus are assigned types, rather than the "Church view" where terms and types are defined simultaneously to produce typed terms.

The set of all λ-terms Λ is built from the set of λ-term variables $\mathcal{V}$ using application and abstraction as specified by the usual grammar $\Lambda ::= \mathcal{V} \mid (\Lambda\,\Lambda) \mid (\lambda\mathcal{V}.\Lambda)$. We use small Roman letters towards the end of the alphabet as metavariables ranging over $\mathcal{V}$ and capital Roman letters as metavariables ranging over Λ. When writing λ-terms, application associates to the left so that $MNP \equiv (MN)P$. The scope of "$\lambda x.$" extends as far to the right as possible.

As usual, $\mathrm{FV}(M)$ and $\mathrm{BV}(M)$ denote the free and bound variables of a λ-term M. By $M[x{:=}N]$ we mean the result of substituting N for all free occurrences of x, renaming bound variables in M to avoid capturing free variables of N. We will sometimes use this substitution notation on subterms when we intend free variables to be captured; we will distinguish this intention by the proper use of parentheses, e.g. in $\lambda x.(N[y := x])$ we intend for the substituted occurrences of x to be captured by the binding. A context $C[\quad]$ is a λ-term with a hole and if M is a λ-term then $C[M]$ denotes the result of inserting M into the hole in $C[\quad]$, *including* the capture of free variables in M by the bound variables of $C[\quad]$. We denote that N is a subterm of M (possibly M itself) by $N \subset M$. We assume at all times that every λ-term M obeys the restriction that no variable is bound more than once and no variable occurs both bound and free in M. The symbol $\mathbf{K}$ denotes the standard combinator $(\lambda x.\lambda y.x)$ and the symbol $\mathbf{I}$ denotes $(\lambda x.x)$.

The set of all types $\mathbb{T}$ is built from the set of type variables $\mathbb{V}$ using two type constructors specified by the grammar $\mathbb{T} ::= \mathbb{V} \mid (\mathbb{T} \to \mathbb{T}) \mid (\forall\mathbb{V}.\mathbb{T})$. A type is therefore either

$$\begin{array}{lll}
\text{VAR} & A \vdash x : \sigma & A(x) = \sigma \\[2ex]
\text{APP} & \dfrac{A \vdash M : \sigma \rightarrow \tau, \qquad A \vdash N : \sigma}{A \vdash (M\ N) : \tau} & \\[2ex]
\text{ABS} & \dfrac{A \cup \{x : \sigma\} \vdash M : \tau}{A \vdash (\lambda x.M) : \sigma \rightarrow \tau} & \\[2ex]
\text{INST} & \dfrac{A \vdash M : \forall\alpha.\sigma}{A \vdash M : \sigma[\alpha := \tau]} & \\[2ex]
\text{GEN} & \dfrac{A \vdash M : \sigma}{A \vdash M : \forall\alpha.\sigma} & \alpha \notin \mathrm{FTV}(A)
\end{array}$$

Figure 1: Inference Rules of System **F** and Λ_k.

a type variable or a $\rightarrow$-*type* or a $\forall$-*type*. We use small Greek letters from the beginning of the alphabet (e.g. α and β) as metavariables over $\mathbb{V}$ and small Greek letters towards the end of the alphabet (e.g. σ and τ) as metavariables over $\mathbb{T}$. When writing types, the arrows associate to the right so that $\sigma \rightarrow \tau \rightarrow \rho = \sigma \rightarrow (\tau \rightarrow \rho)$. We use the same scoping convention for "$\forall$" as we do for "λ". $\mathrm{FTV}(\tau)$ and $\mathrm{BTV}(\tau)$ denote the free and bound type variables of type τ, respectively. We give the notation $\sigma[\alpha := \tau]$ the same meaning for types that it has for λ-terms. We write $\sigma \preceq \tau$ to indicate that σ can be instantiated to τ, i.e. $\sigma = \forall\vec{\alpha}.\rho$ and there exist types $\vec{\chi}$ such that $\rho[\vec{\alpha} := \vec{\chi}] = \tau$. "$\preceq^0$" denotes that the types $\vec{\chi}$ in the substitution contain no quantifiers. We write $\bot$ to denote the type $\forall\alpha.\alpha$.

We have several conventions about how quantifiers in types are treated. α-conversion of types and reordering of adjacent quantifiers is allowed at any time. For example, we consider the types $\forall\alpha.\forall\beta.\alpha \rightarrow \beta$, $\forall\beta.\forall\alpha.\beta \rightarrow \alpha$, and $\forall\beta.\forall\alpha.\alpha \rightarrow \beta$ to all be equal. Using α-conversion we assume that no variable is bound more than once in any type, that the bound type variables of any two type instances are disjoint, and that all bound type variables of any type instance are disjoint from the free type variables of another type instance. If $\sigma = \forall\alpha.\tau$ and $\alpha \notin \mathrm{FTV}(\tau)$, we say that "$\forall\alpha$" is a *redundant* quantifier. We assume types do not contain redundant quantifiers.

We define a notation for specifying many quantifiers concisely. For type σ and set of type variables $\mathbb{X} \subseteq \mathrm{FTV}(\sigma)$, the shorthand notation $\forall\mathbb{X}.\sigma$ is defined so that $\forall\varnothing.\sigma = \sigma$ and $\forall(\mathbb{X} \cup \{\alpha\}).\sigma = \forall\alpha.\forall(\mathbb{X} - \{\alpha\}).\sigma$. This defines just one type because we assume the order of quantifiers does not distinguish two types. We may use $\vec{\alpha}$ to stand for a sequence of type variables $\alpha_1, \ldots, \alpha_n$. We allow $\vec{\alpha}$ to be treated as a set or as a comma-separated sequence as is most convenient, so $\forall\vec{\alpha}.\sigma$ has the expected meaning. The notation $\forall.\sigma$ means $\forall(\mathrm{FTV}(\sigma)).\sigma$.

To define System Λ_k, we will use the following inductive stratification of types by *rank*. First define $\mathbb{R}(0)$ as the set of open types, i.e. types not mentioning the symbol "$\forall$". Then, for all $k \geq 0$, define $\mathbb{R}(k+1)$ by the grammar

$$\mathbb{R}(k+1) ::= \mathbb{R}(k) \mid (\,\mathbb{R}(k) \rightarrow \mathbb{R}(k+1)\,) \mid (\,\forall\mathbb{V}.\mathbb{R}(k+1)\,)$$

We say that $\mathbb{R}(k)$ is the set of types of *rank k*. For example, $\forall\alpha.(\alpha \rightarrow \forall\beta.(\alpha \rightarrow \beta))$ is a type of rank 1 and $(\forall\alpha.(\alpha \rightarrow \alpha)) \rightarrow \forall\beta.\beta$ is a type of rank 2 but not of rank 1. Our definition of rank is exactly the same as the notion of rank introduced by Leivant [Lei83]. Since $\mathbb{R}(k) \subseteq \mathbb{R}(k+1)$ it follows that if a type σ is of rank k, then it also belongs to every rank $n \geq k$. Observe that the result of the substitution $\sigma[\alpha := \tau]$ may not belong to the same ranks to which σ belongs. The resulting rank depends on the rank of τ and how deep in the negative (left-side) scope of "$\rightarrow$" the free occurrences of α in σ are.

$$\begin{array}{lcr}
\text{INST}^- & \dfrac{A \vdash M : \forall\alpha.\sigma}{A \vdash M : \sigma[\alpha := \tau]} & \tau \in \mathbb{S}(0)
\end{array}$$

Figure 2: INST^-: Replacement for INST in Λ_2^-.

To define Λ_2^-, we will define the set $\mathbb{S} \subset \mathbb{T}$ of *restricted types* in which quantifiers can not occur immediately to the right of the arrow "$\rightarrow$". The set $\mathbb{S}$ is defined by the two grammar productions:

$$\begin{aligned}
\mathbb{S} &::= \mathbb{S}' \mid (\forall\mathbb{V}.\mathbb{S}) \\
\mathbb{S}' &::= \mathbb{V} \mid (\mathbb{S} \rightarrow \mathbb{S}')
\end{aligned}$$

The notation $\mathbb{S}(k)$ is defined to mean $\mathbb{S} \cap \mathbb{R}(k)$ and $\mathbb{S}'(k)$ similarly means $\mathbb{S}' \cap \mathbb{R}(k)$.

An *sequent* is an expression of the form $A \vdash M : \tau$ where A is a type assignment (a finite set $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ associating at most one type σ with each variable x), M a λ-term and τ a type. We say this sequent's *type* is the type $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$ and a sequent's *rank* is the rank of its type. For example, a sequent $A \vdash M : \tau$ is of rank 2 if and only if τ is of rank 2 and all the types assigned by A are of rank 1. $A(x)$ denotes the unique type σ such that that $(x : \sigma) \in A$. $\mathrm{FTV}(A)$ is the set of all free type variables in all of the types assigned by A. The notation $A[\vec{\alpha} := \vec{\chi}]$ denotes a new type assignment A' such that if $A(x) = \sigma$ then $A'(x) = \sigma[\vec{\alpha} := \vec{\chi}]$. We assume that throughout a sequent it is the case that all bound type variables are named distinctly from each other and that the bound and free type variables do not overlap (satisfied by α-conversion).

If $\mathcal{K}$ is a type inference system, then the notation $A \vdash_{\mathcal{K}} M : \tau$ denotes the claim that $A \vdash M : \tau$ is derivable in $\mathcal{K}$.

System **F** is the type system that can derive types for λ-terms using the inference rules presented in Figure 1 with no other restrictions. For every $k \geq 0$, System Λ_k is the restriction of **F** which allows only sequents of rank k to be derived.

Definition 2.1 (System Λ_2^-) We define System Λ_2^- as a restriction of System Λ_2 where the two differences are:

1. In Λ_2^- all sequents must have types in $\mathbb{S}(2)$. Thus, all assigned types are in $\mathbb{S}(1)$ and all derived types are in $\mathbb{S}(2)$.

2. The inference rule INST of Λ_2 is replaced by the rule INST^- described in Figure 2 which forbids instantiation with polymorphic types.

To make this paper more self-contained, we will briefly describe the difference in the types that can be assigned to a λ-term in Λ_2 and Λ_2^-. For this description, let us temporarily suppose that quantifiers introduced into types by the INST rule are marked with the "#" symbol. For example, from the sequent $A \vdash M : \forall\alpha.(\alpha \to \alpha)$, we can derive using INST the sequent $A \vdash M : (\forall^{\#}\beta.\beta) \to (\forall^{\#}\beta.\beta)$. These markers on quantifiers do not affect the behavior of the inference rules; they merely allow us to precisely phrase our description.

Definition 2.2 (Quantifier Shifting) We define a mapping $(\ )^\bullet$ that maps every type τ, where quantifiers in τ might be marked with $\#$, to a restricted type in $\mathbb{S}$. The mapping $(\ )^\bullet$ is defined inductively on the structure of types as follows:

$$\begin{aligned}(\alpha)^\bullet &= \alpha\\ (\sigma \to \tau)^\bullet &= \forall\vec{\alpha}.((\sigma)^\bullet \to \rho)\\ &\qquad \text{where } (\tau)^\bullet = \forall\vec{\alpha}.\rho \text{ and } \rho \text{ is not a } \forall\text{-type}\\ (\forall\alpha.\sigma)^\bullet &= \forall\alpha.(\sigma)^\bullet\\ (\forall^{\#}\alpha.\sigma)^\bullet &= (\sigma)^\bullet\end{aligned}$$

For a type assignment A, we define $(A)^\bullet$ so that for every term variable x in the domain of A it holds that $(A)^\bullet(x) = (A(x))^\bullet$.

Theorem 2.3 (Λ_2 **Has Same Power as** Λ_2^-) *System Λ_2^- types the same set of λ-terms as Λ_2 with very similar types. More precisely, if the claim*

$$A \vdash_{\Lambda_2} M : \tau$$

holds, with the additional assumption that quantifiers introduced by INST are marked, then the claim

$$(A)^\bullet \vdash_{(\Lambda_2^-)} M : (\tau)^\bullet$$

holds as well. Also, every derivation in Λ_2^- is immediately a derivation in Λ_2. Thus, Λ_2 and Λ_2^- type the same set of λ-terms.

Theorem 2.3 is Theorem 9 in [KT92]. Since Λ_2^- is as powerful as Λ_2 and since its restrictions make analysis of type inference easier, we will use it instead of Λ_2 in this paper.

3 System Λ_k Typability Undecidable for $k \geq 3$

In this section, we describe a family of type systems, $\Lambda_k[C_k]$ for each $k \geq 3$, for which typability has already been shown to be undecidable. Then we show that the typability problem for each member $\Lambda_k[C_k]$ in this family of type systems is reducible to the typability problem for the corresponding type system Λ_k, thus proving it undecidable as well.

Section 5 of [KT92] introduces System $\Lambda_k[C_k]$ for each $k \geq 3$. Theorem 30 of the same paper proves that typability is undecidable for $\Lambda_k[C_k]$ for $k \geq 3$. The original definition of $\Lambda_k[C_k]$ defined it based on Λ_k by adding two constants, c and f with predefined types $\phi_{c,k}$ and $\phi_{f,k}$ which depend on k. The types $\phi_{c,k}$ and $\phi_{f,k}$ are defined by a simple induction. Let the type $\tau_0 = \alpha$ and then let $\tau_{k+1} = (\tau_k \to \alpha)$ for $k \geq 0$. Then define $\phi_{c,k} = \forall.(\alpha \to \tau_k)$ and $\phi_{f,k} = \forall.((\alpha \to \alpha) \to \tau_{k-1})$. (The fact that both of the types $\phi_{c,k}$ and $\phi_{f,k}$ belong to $\mathbb{S}(1)$ for every k is irrelevant to the definition of System $\Lambda_k[C_k]$.) As an example, for $k = 3$, the types are

$$\begin{aligned}\phi_{c,3} &= \forall\alpha.(\alpha \to (((\alpha \to \alpha) \to \alpha) \to \alpha))\\ \phi_{f,3} &= \forall\alpha.((\alpha \to \alpha) \to ((\alpha \to \alpha) \to \alpha))\end{aligned}$$

A simple alternate definition of $\Lambda_k[C_k]$ which we use in this paper is to declare that $A \vdash M : \tau$ is derivable in $\Lambda_k[C_k]$ if and only if $A \cup \{c : \phi_{c,k}, f : \phi_{f,k}\} \vdash M : \tau$ is derivable in Λ_k.

Lemma 3.1 ($\Lambda_k[C_k]$ **Reducible to** Λ_k) *For each $k \geq 3$, the problem of typability in the system $\Lambda_k[C_k]$ is reducible to the problem of typability in the system Λ_k. More precisely, for each $k \geq 3$, there is a context $H_k[\ \ ]$ with one hole such that for any type assignment A, the statement:*

$$\exists\tau \in \mathbb{R}(k) \textit{ such that } A \cup \{c : \phi_{c,k}, f : \phi_{f,k}\} \vdash_{\Lambda_k} M : \tau$$

is true if and only if the following statement is true:

$$\exists\sigma \in \mathbb{R}(k) \textit{ such that } A \vdash_{\Lambda_k} H_k[M] : \sigma$$

As an example, the context $H_3[\ \ ]$ with one hole may be constructed as depicted in Figure 3. It can be easily checked that the context in Figure 3 can be typed in Λ_3 and somewhat more tediously checked that in any typing of this context (with any λ-term placed in the hole), the variables c and f must be assigned the types $\phi_{c,3}$ and $\phi_{f,3}$. The methods of [Wel93] may be used in a similar manner to construct contexts $H_4[\ \ ]$, $H_5[\ \ ]$, $H_6[\ \ ]$, etc., each more complicated than the previous one.

Theorem 3.2 (Rank ≥ 3 Typability Undecidable) *For $k \geq 3$, since the problem of typability for $\Lambda_k[C_k]$ is reducible to the same problem for Λ_k, and since typability for $\Lambda_k[C_k]$ is undecidable, it is the case that typability is undecidable for Λ_k.*

4 System $\Lambda_2^{-,*}$

In this section, we observe a number of convenient properties of System Λ_2^-. We then define System $\Lambda_2^{-,*}$ as a restriction of Λ_2^- that embodies these properties and prove that $\Lambda_2^{-,*}$ is equivalent to Λ_2^-.

Definition 4.1 (Active Abstractions) Define, by induction on λ-terms M, the sequence $act(M)$ of *active abstractions* in M:

$$\begin{aligned}act(x) &= \varepsilon \text{ (the empty sequence)}\\ act(\lambda x.M) &= x \cdot act(M)\\ act(MN) &= \begin{cases}\varepsilon & \text{if } act(M) = \varepsilon,\\ x_2 \cdots x_n & \text{if } act(M) = x_1 \cdots x_n \text{ for } n \geq 1.\end{cases}\end{aligned}$$

Observe that, due to our conventions on the naming of bound variables, there are no repetitions of variables in $act(M)$. The sequence $act(M)$ represents outstanding abstractions in M, i.e. those abstractions which have not been "captured" by an application.

Definition 4.2 (Companions) For each application subterm $Q \equiv RS$ in a λ-term M where $act(R) = x \cdots$, there is an abstraction subterm $N \equiv (\lambda x.P)$ within R (possibly N is R itself). In this case, we say that the subterms N and S are *companions*. Specifically, N is the *companion abstraction* and S the *companion argument*. If $N \equiv R$, i.e. $Q \equiv NS$, then we say that they are *adjacent companions*.

$$\begin{aligned}
J_i[\ \ ] &\equiv (\lambda y_i.(\lambda z_i.r(y_i y_i(y_i z_i))))(\lambda x_i.\mathbf{K}x_i(\mathbf{K}(x_i(x_i r))[\ \ ]))(\lambda w_i.w_i w_i) \\
D[\ \ ] &\equiv (\lambda f.r(x_1(f x_1 x_1))(x_2(f x_2 x_2))[\ \ ])(\lambda u.\lambda v.u(v(u(ur)))) \\
E[\ \ ] &\equiv (\lambda t.r(x_1(t x_1(x_1 r)(f x_1)))(x_2(t x_2(x_2 r)(f x_2)))[\ \ ])(\lambda p.\lambda q.\lambda s.\mathbf{K}(p(pq))(p(sp))) \\
G[\ \ ] &\equiv (\lambda c.r(x_1(c(x_1 r)(f x_1)))(x_2(c(x_2 r)(f x_2)))[\ \ ])(tr) \\
H_3[\ \ ] &\equiv \lambda r.J_1[J_2[D[E[G[\ \ ]]]]]
\end{aligned}$$

Figure 3: The Context $H_3[\ ]$ used in Reduction from $\Lambda_3[C_3]$ to Λ_3.

It is the case that adjacent companions are always a β-redex. A set of non-adjacent companions represents a "potential" β-redex in a λ-term whose presence can be detected by simple inspection without β-reduction. Consider a λ-term M with subterms $(\lambda x.P)$ and Q which are companions where $(\lambda x.P)$ is the companion abstraction and Q is the companion argument. In this case, if $(\lambda x.P)$ ever participates in a β-redex after some number of steps of β-reduction, its argument will be Q or Q's β-descendent.

Companions will turn out to have convenient properties in System Λ_2^-.

Definition 4.3 (Abstraction Labelling) For a λ-term M, we define $(M)^\lambda$ as the effect of traversing M and labeling each of its abstraction subterms with an index $i \in \{1, 2, 3\}$, depending on the subterm's position and whether it has companions. $(M)^\lambda$ is defined in terms of an auxiliary function *label* which takes as parameters a λ-term, a set of term variables, and an index. The inductive definition of *label* follows for $i \in \{1, 2, 3\}$:

$$\begin{aligned}
\mathit{label}(x, X, i) &= x \\
\mathit{label}((\lambda x.M), X, i) &= \begin{cases} (\lambda^i x.\,\mathit{label}(M, X, i)) & \text{if } x \in X, \\ (\lambda^1 x.\,\mathit{label}(M, X, i)) & \text{if } x \notin X. \end{cases} \\
\mathit{label}((MN), X, i) &= (\mathit{label}(M, X, i) \cdot \mathit{label}(N, \mathit{act}(N), 3))
\end{aligned}$$

We then finish the definition by specifying that

$$(M)^\lambda = \mathit{label}(M, \mathit{act}(M), 2)$$

Informally, labeling the λ-term M affects each abstraction subterm $(\lambda x.N)$ as follows. If $(\lambda x.N)$ has a companion within M, then it is labelled as $(\lambda^1 x.N)$. If $(\lambda x.N)$ does not have a companion within M and if there is no subterm $P = LR$ of M such that N lies within R (the right subterm), then it is labelled as $(\lambda^2 x.N)$. Otherwise it is labelled as $(\lambda^3 x.N)$.

When dealing with a labelled λ-term M after this point, we will assume that the labeling is the result of the $(\ )^\lambda$ operator and not any arbitrary labeling, i.e. we assume that either $M = (N)^\lambda$ or $M \subset (N)^\lambda$ for some unlabelled λ-term N.

Lemma 4.4 (λ^3 Binds Monomorphically) *The bound variable of a companionless, λ^3-labelled abstraction must be assigned a monomorphic type. More precisely, if $\mathcal{D}$ is a derivation in Λ_2^- that types the λ-term M, and there is an abstraction subterm $(\lambda x.N)$ in M, and there is a subterm (PQ) in M such that x appears in $\mathit{act}(Q)$ (i.e. labelling would produce $(\lambda^3 x.N)$), then for every sequent $A \cup \{x : \sigma\} \vdash N : \tau$ in $\mathcal{D}$ it is the case that $\sigma \in \mathbb{S}(0)$.*

Lemma 4.5 (Free Type Variable Substitution) *If $\mathcal{D}$ is a derivation in Λ_2^- ending with the sequent $A \vdash M : \tau$, then for any type variable substitution $[\vec{\alpha} := \vec{\chi}]$, it is the case that there is a derivation $\mathcal{D}'$ in Λ_2^- ending with the sequent $A[\vec{\alpha} := \vec{\chi}] \vdash M : \tau[\vec{\alpha} := \vec{\chi}]$ and, furthermore, $\mathcal{D}$ and $\mathcal{D}'$ are of the same length and there is a one-to-one correspondence between rule applications in both derivations.*

Lemma 4.5 is used by Lemma 4.6. For Lemma 4.6, let us temporarily suppose that quantifiers introduced into types by the GEN rule are marked with the "$\flat$" symbol. For example, from the sequent $A \vdash M : \tau$ where $\alpha \notin \mathrm{FTV}(A)$ we can derive using GEN the sequent $A \vdash M : \forall^\flat \alpha.\tau$. These markers on quantifiers do not affect the behavior of the inference rules; they merely allow us to precisely phrase the claim of the lemma.

Lemma 4.6 (GEN Quantifiers Not Instantiated) *We may freely assume that quantifiers introduced by* GEN *are never instantiated. More precisely, if $\mathcal{D}$ is a derivation in Λ_2^- ending with the sequent $A \vdash M : \tau$, then there is a derivation $\mathcal{D}'$ in Λ_2^- ending with the same sequent such that in $\mathcal{D}'$ there is no use of the* INST *rule whose premise is a sequent of the form $B \vdash N : \forall^\flat \alpha.\rho$.*

Lemma 4.7 (Outermost Quantifiers Only at Companion Arguments) *The only proper subterms of a λ-term for which the final derived type may be a $\forall$-type are companion arguments. More precisely, if $\mathcal{D}$ is a derivation in Λ_2^- that types the λ-term M, and if $\mathcal{D}$ includes the sequent $A \vdash N : \forall\alpha.\tau$, and if there are no subsequent sequents in $\mathcal{D}$ for the same occurrence of the subterm N, then either $N \equiv M$ or this occurrence of N is the argument subterm of a subterm (PN) in M where $\mathit{act}(P) \neq \varepsilon$.*

Lemma 4.8 results from Lemmas 4.6 and 4.7.

Lemma 4.8 (GEN Only at Companion Arguments) *The only proper subterms of a λ-term for which the* GEN *rule may be used are companion arguments. More precisely, if $\mathcal{D}$ is a derivation in Λ_2^- that types the λ-term M, and if $\mathcal{D}$ includes the sequent $A \vdash N : \forall\alpha.\tau$ as a consequence of the* GEN *rule, and if $N \not\equiv M$, then N is a companion argument.*

Lemma 4.9 (INST Only at Variables) *We may freely assume that all uses of the* INST *rule occur at the leaves of the derivation (viewing the derivation as a tree). More precisely, if $\mathcal{D}$ is a derivation in Λ_2^- ending with the sequent $A \vdash M : \tau$, then there is a derivation $\mathcal{D}'$ in Λ_2^- ending with the same sequent such that if the sequent $B \vdash N : \sigma$ in $\mathcal{D}'$ is the consequence of the* INST *rule, then N is a term variable.*

VAR*	$A \vdash x : \forall\vec{\alpha}.\tau$	$A(x) \preceq^0 \tau$,	$\tau \in \mathbb{S}(0)$,		$\vec{\alpha} \notin \mathrm{FTV}(A)$
APP*	$\dfrac{A \vdash M : \sigma \to \tau, \qquad A \vdash N : \sigma}{A \vdash (M\ N) : \forall\vec{\alpha}.\tau}$	$\sigma \in \mathbb{S}(0)$,	$\tau \in \mathbb{S}'(2)$,	$act(M) = \varepsilon$,	$\vec{\alpha} \notin \mathrm{FTV}(A)$
APP$^{*,+}$	$\dfrac{A \vdash M : \sigma \to \tau, \qquad A \vdash N : \sigma}{A \vdash (M\ N) : \forall\vec{\alpha}.\tau}$	$\sigma \in \mathbb{S}(1)$,	$\tau \in \mathbb{S}'(2)$,	$act(M) \neq \varepsilon$,	$\vec{\alpha} \notin \mathrm{FTV}(A)$
ABS^{*,1,2}	$\dfrac{A \cup \{x : \sigma\} \vdash M : \tau}{A \vdash (\lambda^i x.M) : \forall\vec{\alpha}.(\sigma \to \tau)}$	$\sigma \in \mathbb{S}(1)$,	$\tau \in \mathbb{S}(0)$,	$i \in \{1, 2\}$,	$\vec{\alpha} \notin \mathrm{FTV}(A)$
ABS^{*,3}	$\dfrac{A \cup \{x : \sigma\} \vdash M : \tau}{A \vdash (\lambda^3 x.M) : \forall\vec{\alpha}.(\sigma \to \tau)}$	$\sigma \in \mathbb{S}(0)$,	$\tau \in \mathbb{S}(0)$,		$\vec{\alpha} \notin \mathrm{FTV}(A)$

Figure 4: Inference Rules of System $\Lambda_2^{-,*}$.

Definition 4.10 (System $\Lambda_2^{-,*}$) The new System $\Lambda_2^{-,*}$ formally includes the restrictions on Λ_2^- proven by the previous lemmas in a type system. The inference rules for $\Lambda_2^{-,*}$ are in Figure 4. As in Λ_2^-, all sequents are required to be of rank 2, i.e. assigned types must be in $\mathbb{S}(1)$ and derived types must be in $\mathbb{S}(2)$.

Theorem 4.11 ($\Lambda_2^{-,*}$ Equivalent to Λ_2^-) *Every Λ_2^- typing is equivalent to a $\Lambda_2^{-,*}$ typing and vice versa. More precisely, the claim:*

$$A \vdash_{(\Lambda_2^-)} M : \tau$$

holds if and only if the following claim holds:

$$A \vdash_{(\Lambda_2^{-,*})} (M)^\lambda : \tau$$

5 θ-Reduction and System $\Lambda_2^{-,*,\theta}$

In this section, we define a new notion of reduction and then use it to reduce System $\Lambda_2^{-,*}$ typability to an even more restricted type discipline, System $\Lambda_2^{-,*,\theta}$.

Definition 5.1 (θ-Reduction) We define 4 notions of reduction denoted θ_1, θ_2, θ_3, and θ_4 which will transform a labelled λ-term $(M)^\lambda$ in a useful way. These transformations are defined as follows:

- θ_1 transforms a subterm of the form $(((\lambda^1 x.N)P)Q)$ to $((\lambda^1 x.NQ)P)$.
- θ_2 transforms a subterm $(\lambda^3 x.(\lambda^1 y.N)P)$ into the form $((\lambda^1 v.\lambda^3 x.(N[y{:=}vx]))(\lambda^3 w.(P[x{:=}w])))$, where v and w are fresh variables.
- θ_3 transforms a subterm of the form $(N((\lambda^1 x.P)Q))$ to $((\lambda^1 x.NP)Q)$.
- θ_4 transforms a subterm of the form $((\lambda^1 x.(\lambda^2 y.N))P)$ to $(\lambda^2 y.((\lambda^1 x.N)P))$.

Capture of free variables in θ_1, θ_3, and θ_4 does not occur due to our assumption that all bound variables are named distinctly from all free variables. θ_1, θ_3, and θ_4 affect subterms that are applications, while θ_2 is applied to subterms that are abstractions. When λ-terms are viewed as trees, θ_1, θ_2, and θ_3 can be seen to have the effect of hoisting β-redexes higher in the transformed term, while θ_4 has the effect of raising an abstraction above a β-redex. In section 6, we will use properties of these transformations to prove that a typability problem is reducible to acyclic semi-unification.

We use the notation θ_i where $i \in \{1, 2, 3, 4\}$ to stand for one of θ_1, θ_2, θ_3, or θ_4. We define $\theta = \theta_1 \cup \theta_2 \cup \theta_3 \cup \theta_4$. Since these transformations are all notions of reduction, the notations $\to_{\theta_1}$, $\to_{\theta_2}$, $\to_\theta$, etc., have the expected meaning.

We say that a term is in *θ-normal form* if it has no θ-redexes. A θ-normal form of M is a λ-term N in θ-normal form such that $M \twoheadrightarrow_\theta N$. A λ-term may have more than one θ-normal form, e.g. the λ-term $(((\lambda x.M)N)((\lambda y.P)Q))$ has two θ-normal forms: the λ-term $((\lambda x.(\lambda y.MP)Q)N)$ and the λ-term $((\lambda y.(\lambda x.MP)N)Q)$.

We now describe some useful properties of θ-reduction.

Lemma 5.2 (Shape of θ-Normal Forms) *Let M be in θ-normal form. Then M is of the form*

$$\begin{aligned} M \;\equiv\; & \lambda^2 x_1.\lambda^2 x_2.\ldots\lambda^2 x_m. \\ & (\lambda^1 y_1.(\lambda^1 y_2.(\ldots((\lambda^1 y_n.T_{n+1})T_n)\ldots))T_2)T_1 \end{aligned}$$

where $m, n \geq 0$ and where for $1 \leq i \leq n+1$ each subterm T_i is in β-normal form and any abtractions within T_i are λ^3-labelled.

Observe that in a θ-normal form all λ^1-labelled abstractions belong to β-redexes, i.e. there are no non-adjacent companions. The λ-term M detailed in Lemma 5.2 can also be viewed as the following **ML** term:

$$\begin{array}{c} \textbf{fn } x_1 \Rightarrow \textbf{fn } x_2 \Rightarrow \cdots \Rightarrow \textbf{fn } x_m \Rightarrow \\ \textbf{let } y_1 = T_1 \textbf{ in let } y_2 = T_2 \textbf{ in } \cdots \\ \textbf{let } y_n = T_n \textbf{ in } T_{n+1} \end{array}$$

Lemma 5.3 (β-Equivalence Preserved) *θ_1, θ_2, θ_3, and θ_4 always transform a λ-term M into a β-equivalent λ-term N, i.e. if $M \to_\theta N$, then $M =_\beta N$.*

To prove that any θ-reduction terminates, we establish a metric on λ-terms and we then show that θ-reduction strictly decreases this metric.

Definition 5.4 (Distance from θ-Normal Form) We will define a function from labelled λ-terms to natural numbers using the following components. In the following definitions, we presume that each subterm of a λ-term is somehow distinctly indexed, so that otherwise identical subterms in different positions are distinguished. This is important so that the desired answers are produced when counting the size of a set of subterms and when asking whether one subterm is a subterm of another subterm.

Let A (for "ancestors") be a function that takes a labelled λ-term M and a subterm N within M, and returns the set of all subterms of M which contain N (including M and N). Let β (for "β-redexes") be a function that takes a λ-term M and returns the set of all of the subterms of M that are either β-redexes or are the function of a β-redex. Let λ^i for each $i \in \{1, 2, 3\}$ be a function that takes a λ-term M and returns the set of all subterms of M that are λ^i-labelled abstractions.

Now define three metric functions δ_1, δ_2, and δ_3 which are used to measure the distance of a λ-term from θ-normal form. δ_1 takes a λ-term M and a subterm N and returns the number of subterms of M that contain N that are neither β-redexes, the function of a β-redex, nor a λ^2-labelled abstraction:

$$\delta_1(M, N) = \left| A(M, N) - \beta(M) - \lambda^2(M) \right|$$

δ_2 takes a λ-term M and a subterm N and returns the number of application subterms in M that contain N as a subterm of their right subterm:

$$\delta_2(M, N) = \left| \{ P \mid P \in A(M, N),\ P \equiv QR,\ R \in A(M, N) \} \right|$$

δ_3 takes a λ-term M and a subterm N and returns the number of subterms in M that are λ^2-labelled abstraction and do not properly contain N:

$$\delta_3(M, N) = \left| \lambda^2(M) - \{N\} - A(M, N) \right|$$

Now use δ_1, δ_2, and δ_3 to define the metric function d to measure how far a λ-term is from θ-normal form. Define d as follows:

$$d(M) = \sum_{N \in \lambda^1(M)} \delta_1(M, N) + \delta_2(M, N) + \sum_{N \subset M} \delta_3(M, N)$$

Note that δ_1 and δ_2 are applied just to the λ^1-labelled abstractions in M, i.e. the abstractions that have companions, but δ_3 is applied to all subterms of M.

Lemma 5.5 (θ-Reduction Terminates) *θ-reduction always terminates (strongly normalizes). More precisely, if $M \rightarrow_{\theta_i} N$, then $d(M) > d(N)$. Furthermore, for a λ-term M, it holds that $d(M) \in O(|M|^2)$, so it takes $O(|M|^2)$ steps of θ-reduction to reach θ-normal form.*

Lemma 5.6 (β-Redex Binds Partly Closed Type) *We may freely assume that for the type σ assigned to the bound variable of a λ^1-abstraction which is the function of a β-redex, it is the case that any free type variables in σ must also be free somewhere else in the type assignment. More precisely, if $\mathcal{D}$ is a derivation in $\Lambda_2^{-,*}$ containing the sequent $A \vdash (\lambda^1 x.M)N : \tau$ which is derived from the earlier sequents $A \cup \{x : \sigma\} \vdash M : \tau$ and $A \vdash N : \sigma$, then there is also a derivation $\mathcal{D}'$ in $\Lambda_2^{-,*}$ containing the same sequent but in which the sequent is derived instead from earlier sequents $A \cup \{x : \sigma'\} \vdash M : \tau$ and $A \vdash N : \sigma'$ where $\sigma' \equiv \forall(\mathrm{FTV}(\sigma) - \mathrm{FTV}(A)).\sigma$.*

Lemma 5.6 is used by Lemma 5.7.

Lemma 5.7 ($\Lambda_2^{-,*}$ Typings Preserved) *If θ_1, θ_2, θ_3, or θ_4 transform M into N in one step, then with any particular type assignment, both M and N are typable with the same types in $\Lambda_2^{-,*}$. In other words, if $M \rightarrow_\theta N$, then in $\Lambda_2^{-,*}$ it holds that $A \vdash M : \tau$ is derivable if and only if $A \vdash N : \tau$ is derivable. As a result, $A \vdash_{(\Lambda_2^{-,*})} M : \tau$ is true if and only if $A \vdash_{(\Lambda_2^{-,*})} \theta\text{-}nf(M) : \tau$ is true.*

Lemma 5.8 (Active Abstractions Preserved) *The set of active abstractions of a λ-term is preserved by θ-reduction. As a result, $act(\theta\text{-}nf((M)^\lambda)) = act(M)$.*

Lemma 5.9 (Shape of Derivable Types) *In $\Lambda_2^{-,*}$, if $A \vdash M : \rho$ is derivable and $|act(M)| = n$, then*

$$\rho = \forall\vec{\alpha}.\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \tau$$

where $\vec{\sigma} \in \mathbb{S}(1)$ and $\tau \in \mathbb{S}(0)$.

Lemma 5.9 was proven in [KT92].

Lemma 5.10 (λ^2 Can Bind Closed Type) *We can always assign a closed type or even the type $\bot = \forall\alpha.\alpha$ to the bound variable of a companionless, λ^2-labelled abstraction without affecting the whole λ-term's typability. More precisely, if $\mathcal{D}$ is a typing in $\Lambda_2^{-,*}$ of the λ-term M ending with the sequent*

$$A \vdash M : \forall\vec{\alpha}.\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \tau$$

where $|act(M)| = n$, then there is a typing $\mathcal{D}'$ ending with the sequent

$$A \vdash M : \forall\vec{\beta}.(\forall.\sigma_1) \rightarrow \ldots \rightarrow (\forall.\sigma_n) \rightarrow \tau$$

where $\vec{\beta} = \vec{\alpha} - (\mathrm{FTV}(\vec{\sigma}) - \mathrm{FTV}(\tau))$ and there is also a derivation $\mathcal{D}''$ ending with the sequent

$$A \vdash M : \forall\vec{\beta}.\bot \rightarrow \ldots \rightarrow \bot \rightarrow \tau$$

Lemma 5.11 (λ^1 Can Bind Closed Type in θ-nf) *Provided the final type assignment in a derivation assigns closed types to all free variables, and provided that every λ^2-abstraction binds a variable with a closed type, and provided we are typing a λ-term in θ-normal form, then we can assign closed types to the bound variables of every λ^1-labelled abstraction without affecting the whole λ-term's typability.*

Definition 5.12 (System $\Lambda_2^{-,*,\theta}$) The new System $\Lambda_2^{-,*,\theta}$ takes advantage of the typing properties of λ-terms in θ-normal form in $\Lambda_2^{-,*}$. System $\Lambda_2^{-,*,\theta}$ is intended to be used only for θ-normal forms; its behavior on other λ-terms has not been investigated. The inference rules for $\Lambda_2^{-,*,\theta}$ are presented in Figure 5. As for $\Lambda_2^{-,*}$, all sequents are required to be of rank 2, i.e. assigned types must be in $\mathbb{S}(1)$ and derived types must be in $\mathbb{S}(2)$. We adopt the convention that the final type assignment of any typing in $\Lambda_2^{-,*,\theta}$ must assign closed (universally polymorphic) types to every free variable, otherwise the derivation is considered incomplete.

VAR^{θ}	$A \vdash x : \tau$	$A(x) \preceq^0 \tau, \quad \tau \in \mathbb{S}(0)$
APP^{θ}	$\dfrac{A \vdash M : \sigma \to \tau, \qquad A \vdash N : \sigma}{A \vdash (M\,N) : \tau}$	$\sigma, \tau \in \mathbb{S}(0), \quad M$ not abstraction
LET^{θ}	$\dfrac{A \cup \{x : \forall.\sigma\} \vdash M : \tau, \qquad A \vdash N : \sigma}{A \vdash ((\lambda^1 x.M)\,N) : \tau}$	$\sigma, \tau \in \mathbb{S}(0), \quad \mathrm{FTV}(A) = \varnothing$
$\mathrm{ABS}^{\theta,2}$	$\dfrac{A \cup \{x : \forall.\sigma\} \vdash M : \tau}{A \vdash (\lambda^2 x.M) : (\forall.\sigma) \to \tau}$	$\sigma \in \mathbb{S}(0), \quad \tau \in \mathbb{S}'(2)$
$\mathrm{ABS}^{\theta,3}$	$\dfrac{A \cup \{x : \sigma\} \vdash M : \tau}{A \vdash (\lambda^3 x.M) : \sigma \to \tau}$	$\sigma, \tau \in \mathbb{S}(0)$

Figure 5: Inference Rules of System $\Lambda_2^{-,*,\theta}$.

Theorem 5.13 ($\Lambda_2^{-,*}$ **Reducible to** $\Lambda_2^{-,*,\theta}$) *Typability and type inference in* $\Lambda_2^{-,*}$ *are reducible to the same problems in* $\Lambda_2^{-,*,\theta}$. *For a labelled* λ*-term* M *where* $|act(M)| = n$, *if*

$$A \vdash_{(\Lambda_2^{-,*})} M : \forall\vec{\alpha}.\sigma_1 \to \cdots \to \sigma_n \to \tau$$

is true, then using the type assignment B *such that for* $x \in \mathrm{FV}(M)$ *it behaves so that* $B(x) = \forall.A(x)$, *it is the case that*

$$B \vdash_{(\Lambda_2^{-,*,\theta})} \theta\text{-}nf(M) : (\forall.\sigma_1) \to \cdots \to (\forall.\sigma_n) \to \tau$$

and using the type assignment C *that maps all free variables to type* $\perp$ *it is the case that*

$$C \vdash_{(\Lambda_2^{-,*,\theta})} \theta\text{-}nf(M) : \perp \to \cdots \to \perp \to \tau$$

Also, every derivation in $\Lambda_2^{-,*,\theta}$ *is immediately a derivation in* $\Lambda_2^{-,*}$, *so if*

$$A \vdash_{(\Lambda_2^{-,*,\theta})} \theta\text{-}nf(M) : \rho$$

is true, then

$$A \vdash_{(\Lambda_2^{-,*})} M : \rho$$

must be true as well.

6 System $\Lambda_2^{-,*,\theta}$ Type Inference Reducible to ASUP

In this section, we define acyclic semi-unification, give an algorithm for solving this problem, and develop a construction for reducing the problem of typability in System $\Lambda_2^{-,*,\theta}$ to acyclic semi-unification.

Definition 6.1 (Semi-Unification (SUP)) For convenience, we define semi-unification using the set of open types $\mathbb{R}(0)$ as the set of algebraic terms $\mathcal{T}$. Let $X = \mathbb{V}$ denote the set of term variables to emphasize their use in algebraic terms as opposed to types. Although the members of $\mathcal{T}$ are also types, we will refer to them as terms when using them in semi-unification. A *substitution* is a function $S : X \to \mathcal{T}$ that differs from the identity on only finitely many variables. Every substitution extends in a natural way to a "$\to$"-homomorphism $S : \mathcal{T} \to \mathcal{T}$ so that $S(\sigma \to \tau) = S(\sigma) \to S(\tau)$. An *instance* Γ of *semi-unification* is a finite set of pairs (called inequalities) in $\mathcal{T} \times \mathcal{T}$. Each such pair is written as $\tau \leq \mu$ where $\tau, \mu \in \mathcal{T}$. A substitution S is a *solution* of instance $\Gamma = \{\tau_1 \leq \mu_1, \ldots, \tau_n \leq \mu_n\}$ if and only if there exist substitutions $R_1, \ldots, R_n$ such that:

$$R_1(S(\tau_1)) = S(\mu_1)\ ,\ \ldots\ ,\ R_n(S(\tau_n)) = S(\mu_n)$$

The *semi-unification problem* (henceforth abbreviated SUP) is the problem of deciding, for a SUP instance Γ, whether Γ has a solution.

Definition 6.2 (Acyclic Semi-Unification (ASUP)) An instance Γ of semi-unification is *acyclic* if it can be organized as follows. There are $n+1$ disjoint sets of variables, $V_0, \ldots, V_n$, for some $n \geq 1$, such that the inequalities of Γ can be placed into n columns:

$$\begin{array}{cccc}
\tau^{1,1} \leq \mu^{1,1} & \tau^{2,1} \leq \mu^{2,1} & \cdots & \tau^{n,1} \leq \mu^{n,1} \\
\tau^{1,2} \leq \mu^{1,2} & \tau^{2,2} \leq \mu^{2,2} & \cdots & \tau^{n,2} \leq \mu^{n,2} \\
\vdots & \vdots & & \vdots \\
\tau^{1,r_1} \leq \mu^{1,r_1} & \tau^{2,r_2} \leq \mu^{2,r_2} & \cdots & \tau^{n,r_n} \leq \mu^{n,r_n}
\end{array}$$

where for $0 \leq i \leq n$:

$$V_i = \{\ \alpha \mid \exists j.\, \alpha \in \mathrm{FTV}(\tau^{i+1,j}) \text{ or } \alpha \in \mathrm{FTV}(\mu^{i,j})\ \}$$

The *acyclic semi-unification problem* (henceforth abbreviated ASUP) is the problem of deciding, for an ASUP instance Γ, whether Γ has a solution.

Definition 6.3 (Paths in Terms) For an arbitrary algebraic term τ, we define the *left* and *right* subterms of τ, denoted $L(\tau)$ and $R(\tau)$. More precisely, if τ is a variable then $L(\tau)$ and $R(\tau)$ are undefined, otherwise we set $L(\tau^1 \to \tau^2) = \tau^1$ and $R(\tau^1 \to \tau^2) = \tau^2$. If $\Pi \in \{L, R\}^*$, say $\Pi = x_1 x_2 \cdots x_p$, the notation $\Pi(\tau)$ means $x_1(x_2(\cdots(x_p(\tau)\cdots))$. For an arbitrary $\Pi \in \{L, R\}^*$, the subterm $\Pi(\tau)$ is defined provided Π (read from right to left) is a path (from the root to an internal node or to a leaf node) in the binary tree representation of τ.

The following algorithm is an important sub-algorithm of the overall type-inference algorithm for Λ_2.

Algorithm 6.4 (Redex Procedure) We now define a procedure (modified from [KTU93]) to solve instances of ASUP. This procedure repeatedly reduces *redexes* of two kinds and it halts if there are no more redexes or if a conflict is detected that precludes a solution. Each reduction substitutes a term for a variable throughout Γ and the composition of the reductions done so far represents the construction of the solution.

Redex I Reduction: Let $\xi \in X$ and let $\tau' \notin X$ be a term with the property that there is a path $\Pi \in \{L, R\}^*$ and $\tau \leq \mu$ is an inequality of Γ such that:

$$\Pi(\tau) = \tau' \quad \text{and} \quad \Pi(\mu) = \xi$$

The pair of terms $(\xi, T(\tau'))$ where T is a one-to-one substitution that maps all variables in τ' to fresh names is called a *redex I*. Reducing this redex substitutes $T(\tau')$ for all occurrences of ξ throughout Γ.

Redex II Reduction: Let $\xi \in X$ and $\mu' \in \mathcal{T}$ have the property that $\xi \neq \mu'$ and there are paths $\Pi, \Delta, \Sigma \in \{L, R\}^*$ and $\tau \leq \mu$ is an inequality in Γ such that:

$$\begin{array}{rclcrcl} \Pi(\tau) & \in & X & \quad & \Pi(\tau) & = & \Delta(\tau) \\ \Sigma\Pi(\mu) & = & \xi & & \Sigma\Delta(\mu) & = & \mu' \end{array}$$

Such a pair (ξ, μ') is called a *redex II*. Reducing this redex consists of substituting μ' for all occurrences of ξ throughout Γ. However, if there is a path $\Theta \in \{L, R\}^*$ such that $\Theta(\mu') = \xi$, then no solution to Γ is possible, so the procedure halts and outputs the answer that there is no solution if this is detected.

Although the general case of SUP has been proven to be undecidable [KTU93], ASUP has been proven to be decidable and in fact it is DEXPTIME-complete [KTU90] (where DEXPTIME means DTIME($2^{n^{O(1)}}$)). In addition, we have the following result for ASUP.

Lemma 6.5 (Redex Procedure Solves ASUP) *For an instance Γ of* ASUP*, the redex procedure either constructs a solution S to Γ and halts or correctly answers that Γ has no solution and halts. Furthermore, it halts within exponential time.*

We now define another important sub-algorithm of the type-inference algorithm for Λ_2.

Algorithm 6.6 (Constructing Γ_M) To solve the typability and type inference problems for $\Lambda_2^{-,*,\theta}$ for λ-terms in θ-normal form, we construct for a λ-term M an ASUP instance Γ_M. Consider the labelled λ-term M in θ-normal form:

$$\begin{array}{rl} M \equiv & \lambda^2 x_1.\lambda^2 x_2.\ldots\lambda^2 x_m. \\ & (\lambda^1 y_1.(\lambda^1 y_2.(\ldots((\lambda^1 y_n.T_{n+1})T_n)\ldots))T_2)T_1 \end{array}$$

We will adopt the convention that the abstractions in a component T_i for some i bind variables named $z_{i,1}$, $z_{i,2}$, etc., and that the free variables of M are named $w_1, w_2, \ldots, w_p$. By writing the inequality $(\tau \leq_i \mu)$, we assert that the inequality will belong to column i of Γ, which will have $n+2$ columns numbered from 0 through $n+1$. (We omit the proof that the resulting set of inequalities Γ_M is of the correct acyclic form to be an instance of ASUP.) Most of the inequalities will be of a certain special form, so $(\tau \doteq_i \mu)$ denotes the inequality $(\alpha \to \alpha \leq_i \tau \to \mu)$ where α is a fresh variable mentioned in no other term in Γ. This will have the effect of unifying τ and μ as in ordinary first-order unification. We will assume that the subterms of M are indexed so that two otherwise identical subterms in different positions within M will be considered distinct in what follows.

We construct the instance Γ_M of ASUP from the λ-term M as follows. In constructing Γ_M, each subterm $N \subset T_i$ for some i will contribute one inequality, each β-redex $((\lambda^1 y_i.P_i)T_i)$ will contribute one inequality, each variable y_i will contribute $n - i + 1$ inequalities, and each variable x_i or w_i will contribute n inequalities. For each subterm N of T_i for some i, the term variable δ_N will represent the derived type of N. For each bound variable $z_{i,j}$ (which must be monomorphic), the term variable $\gamma_{i,j}$ will represent its assigned type. For each variable x_i (respectively y_i or w_i), which must be assigned a universally polymorphic type, the term variables $\beta^x_{0,i}, \ldots, \beta^x_{n,i}$ (respectively $\beta^y_{i,i}, \ldots, \beta^y_{n,i}$ and $\beta^w_{0,i}, \ldots, \beta^w_{n,i}$) will represent its assigned type.

Now we define the inequalities that will be in Γ_M. For each subterm N of T_i for some i, we add an inequality to Γ_M that will depend on N:

1. For $N \equiv w_j$, we add $(\beta^w_{i-1,j} \leq_i \delta_N)$.
2. For $N \equiv x_j$, we add $(\beta^x_{i-1,j} \leq_i \delta_N)$.
3. For $N \equiv y_j$, we add $(\beta^y_{i-1,j} \leq_i \delta_N)$.
4. For $N \equiv z_{i,j}$, we add $(\gamma_{i,j} \doteq_i \delta_N)$.
5. For $N \equiv (PQ)$, we add $(\delta_P \doteq_i \delta_Q \to \delta_N)$.
6. For $N \equiv (\lambda^3 z_{i,j}.P)$, we add $(\gamma_{i,j} \to \delta_P \doteq_i \delta_N)$.

For each β-redex $((\lambda^1 y_i.P_i)T_i)$, we add $(\beta^y_{i,i} \doteq_i \delta_{T_i})$. For each variable x_j (respectively y_j or w_j) and for $1 \leq i \leq n$ (for y_j require $i \geq j+1$ as well) we add the inequality $(\beta^x_{i-1,j} \leq_i \beta^x_{i,j})$ (respectively $(\beta^y_{i-1,j} \leq_i \beta^y_{i,j})$ or $(\beta^w_{i-1,j} \leq_i \beta^w_{i,j})$).

The only remaining consideration is what types to assign to the λ^2-bound variables $x_1, \ldots, x_m$ and to the free variables $w_1, \ldots, w_p$. If our only concern is whether M can be typed at all, then we can assign the type $\bot$ to these variables, in which case we do not need to add anything more to Γ_M. On the other hand, we may wish to specify more complex assigned types for these variables. Let A be a type assignment whose domain is $\{x_1, \ldots, x_m, w_1, \ldots, w_p\}$ and whose range is in $(\mathbb{S}(1) - \mathbb{S}(0))$. We define $\Gamma_{M,A}$ to be Γ_M with the addition of more inequalities. If $A(x_i)$ (respectively $A(w_i)$) is $\forall.\sigma$ where $\sigma \in \mathbb{S}(0)$, we add $(\beta^x_{0,i} \doteq_0 \sigma)$ (respectively $(\beta^w_{0,i} \doteq_0 \sigma)$).

Theorem 6.7 ($\Lambda_2^{-,*,\theta}$ Reducible to ASUP) *Type inference in $\Lambda_2^{-,*,\theta}$ is reducible to* ASUP*. More precisely, let M be a λ-term in θ-normal form of the shape mentioned in Algorithm 6.6. Let $act(M) = x_1 \ldots x_m$. Let A be a type assignment whose domain is* FV$(M) \cup \{x_1, \ldots, x_m\}$ *and whose range is in $(\mathbb{S}(1) - \mathbb{S}(0))$. Let Γ_M be the* ASUP *instance defined by the algorithm. Let $\delta_{T_{n+1}}$ be the term variable appearing in Γ_M which is mentioned in the algorithm. The following statements are true:*

1. Γ_M has a solution S if and only if M is typable in $\Lambda_2^{-,*,\theta}$. Furthermore, if τ is the type

$$\tau = \bot \to \cdots \to \bot \to (S(\delta_{T_{n+1}}))$$

where the number of "$\bot$" components in τ is m, then τ is a type derivable for M in $\Lambda_2^{-,*,\theta}$.

2. If S be a solution for $\Gamma_{M,A}$, then the type

$$\tau = A(x_1) \to \cdots \to A(x_m) \to (S(\delta_{T_{n+1}}))$$

is a type derivable for M in $\Lambda_2^{-,*,\theta}$.

Algorithm 6.8 (Type Inference for Λ_2) We can finally summarize our type inference algorithm for System Λ_2. If M is typable in Λ_2, then the following procedure will produce a type for it and will otherwise answer that M is not typable in Λ_2:

1. Compute the labelled λ-term $M_1 \equiv (M)^\lambda$.
2. Compute the λ-term $M_2 \equiv \theta\text{-nf}(M_1)$ using θ-reduction.
3. Choose a type assignment A for the free and λ^2-bound variables of M. If $act(M) = x_1 \ldots x_m$, let the domain of A be $\text{FV}(M) \cup \{x_1, \ldots, x_m\}$ and let the range of A be in $(\mathbb{S}(1) - \mathbb{S}(0))$. It is possible to choose the trivial type assignment that assigns $\bot$ to all variables.
4. Compute the ASUP instance $\Gamma_{M_2,A}$ (Algorithm 6.6).
5. Run the redex procedure (Algorithm 6.4) on $\Gamma_{M_2,A}$ to either produce a solution S for $\Gamma_{M_2,A}$ or the answer that $\Gamma_{M_2,A}$ has no solution. In the latter case, halt with the answer that M is not typable in Λ_2 with the assumptions of the type assignment A. If A is the trivial type assignment, then M is not typable at all in Λ_2.
6. Compute and output the type

$$A(x_1) \to \cdots \to A(x_m) \to (S(\delta_{T_{n+1}}))$$

where $\delta_{T_{n+1}}$ is the term variable appearing in Γ_M which is mentioned in Algorithm 6.4.

The reader should observe that Algorithm 6.8 makes no reference to the type systems Λ_2^-, $\Lambda_2^{-,*}$, or $\Lambda_2^{-,*,\theta}$. These type systems are used solely to prove that the output of the algorithm in its final step is a correct result. The final result is a valid typing in $\Lambda_2^{-,*,\theta}$, but it is also immediately a valid typing in $\Lambda_2^{-,*}$, Λ_2^-, and Λ_2 as well (after removing any λ-labelling).

We now analyze the complexity of Algorithm 6.8. The initial stages of computing the labelling $M_1 \equiv (M)^\lambda$, the θ-normal form $M_2 \equiv \theta\text{-nf}(M_1)$, and the ASUP instance $\Gamma_{M_2,A}$ can be done in polynomial time. Algorithm 6.4 solves the ASUP instance $\Gamma_{M_2,A}$ in exponential time. Thus, Algorithm 6.8 takes exponential time. Since Λ_2 typability has been shown to be DEXPTIME-complete [KT92], the algorithm is optimal.

To use System Λ_2 or Λ_2^- in an actual programming language, we will have to take account of constants with constant types, e.g. "**true** : **Bool**". This might seem difficult to do, since the type inference algorithm is based on System $\Lambda_2^{-,*,\theta}$ which requires all types assigned to identifiers to be completely closed (polymorphic). However, the redex procedure for solving ASUP instances can be simply told that certain variables are actually constants (e.g. **Bool**) and not to be changed by substitution. Then the type inference algorithm will work correctly with constants.

7 Principal Typing in System Λ_2

In this section, we first observe that in general there are no principal types for Λ_2. Then we describe the principality of solutions to instances of SUP and ASUP and how this relates to types in Λ_2. Finally, we discuss the weak forms of type principality that exist in Λ_2.

It is easy to observe that principal types do not exist in System Λ_2 in the same sense that they do in **ML**. Consider the identity function, $\mathsf{I} \equiv (\lambda x.x)$. In Λ_2^-, all of the types

$$\begin{aligned}\varphi &= (\forall\alpha.\alpha) \to (\beta \to \beta)\\ \psi &= (\forall\alpha.(\alpha \to \alpha)) \to (\beta \to \beta)\\ \pi &= \forall\alpha.(\alpha \to \alpha)\end{aligned}$$

can be derived for I. (Note that π can not be derived for I in $\Lambda_2^{-,*,\theta}$.) However, there is no type τ derivable for I in Λ_2 such that $\tau \preceq \varphi$, $\tau \preceq \psi$, and $\tau \preceq \pi$. When we consider the full power of Λ_2 and the polymorphic instantiation and types in $(\mathbb{R}(2) - \mathbb{S}(2))$ that it allows, the situation seems even more disconcerting.

We do not currently know of a convenient way to represent all of the possible rank-2 types that can be derived for a λ-term. The types derived by our type inference algorithm are principal in a weak sense. The rest of this section will present what is known about the kind of weak principality of types that exists.

The solutions to instances of SUP and ASUP are principal in a weak sense. For substitutions $S, R : X \to \mathcal{T}$, let the notation $S \sqsubseteq R$ mean that there exists some substitution $S' : X \to \mathcal{T}$ such that for all term variables α in the domain of S it holds that $R(\alpha) = S'(S(\alpha))$.

Lemma 7.1 (Principal SUP Solution) *If Γ is an instance of* SUP*, then Γ has a principal solution S such that for every solution R of Γ it is the case that $S \sqsubseteq R$.*

Lemma 7.1 is Proposition 3 in [KTU93].

Lemma 7.2 (Principal ASUP Solution) *Suppose Γ is an instance of* ASUP *with n columns. There are therefore $n+1$ disjoint sets of variables occurring in Γ, which we call V_0, V_1, ..., V_n, satisfying the property that for every inequality $(\tau \leq \mu)$, if $\alpha \in V_i$ occurs in τ, then all the term variables in τ also belong to V_i and all of the term variables in μ belong to V_{i+1}. Let $V = V_0 \cup \cdots \cup V_n$. For a substitution $T : V \to \mathcal{T}$, let the notation $[T]_i$ denote the restriction of T to the domain of V_i. Suppose S is the principal solution of Γ according to Lemma 7.1. Then the conclusion of this lemma is that for any substitution $P : V_i \to \mathcal{T}$ such that $[S]_i \sqsubseteq P$, there is a substitution $R : V \to \mathcal{T}$ such that:*

1. R is a solution of Γ.

2. $[R]_i = P$.

Now, from Lemma 7.2 follows the weak principal typing property for System $\Lambda_2^{-,*,\theta}$.

Theorem 7.3 (Weak Principal Types) *Consider the type computed by Algorithm 6.8 from $\Gamma_{M,A}$ for λ-term M relative to type assignment A for the free and λ^2-bound variables of M:*

$$A(x_1) \to \cdots \to A(x_m) \to (S(\delta_{T_{n+1}}))$$

For any substitution $P : X \to \mathcal{T}$, the following type is derivable for M relative to A:

$$A(x_1) \to \cdots \to A(x_m) \to (P(S(\delta_{T_{n+1}})))$$

Theorem 7.3 holds since S is a solution for $\Gamma_{M,A}$ and $S \sqsubseteq S \circ P$, so there is a solution R such that $[R]_{n+1} = S \circ P$ (where V_{n+1} is the rightmost set of variables in $\Gamma_{M,A}$).

Now consider the types inferred by Algorithm 6.8 for a λ-term M under various restrictions. Suppose M has no λ^2-labelled abstractions and no free variables. In that case, the computed type is exactly $S(\delta_{T_{n+1}})$. By Theorem 7.3, any substitution instance of this type is also a valid type for M. Thus, in this case there is the same sort of strong principality of types that there is in **ML**.

Now consider various cases where M has either λ^2-bound variables or free variables or both.

Suppose in type inference we decide to assign the type $\bot$ to all free and λ^2-bound variables, which provides the maximum possibilities for M to be typed. In this case, the final sequent of the typing will look like this:

$$\{w_1 : \bot, \ldots, w_p : \bot\} \vdash M : \bot \to \cdots \to \bot \to (S(\delta_{T_{n+1}}))$$

The rightmost component of the type, $S(\delta_{T_{n+1}})$, can be replaced with any substitution instance of it. However, since there are no closed λ-terms in Λ_2 for which the type $\bot$ can be derived, this typing does not help us know with what other λ-terms M can be combined. Although assigning $\bot$ to all free and λ^2-bound variables allows us to tell whether a λ-term is typable at all, it seems unlikely to be useful in practice.

A problem with the type inference algorithm and $\Lambda_2^{-,*,\theta}$ is that certain important and natural types will not be assigned to λ-terms unless a trick is used. For example, the type inference algorithm will not derive the type $\forall\alpha.(\alpha \to \alpha)$ for the λ-term $\mathbf{I} \equiv (\lambda x.x)$. The reason for this is that after labelling, the λ-term is $(\lambda^2 x.x)$ and the type assigned to λ^2-bound variables is required to be closed. The type inference algorithm can assign the type $\forall\alpha.(\alpha \to \alpha)$ to the λ-term $(\lambda^3 x.x)$, which is the same λ-term labelled differently. This is not a problem in typing a real program, because whenever the type $\forall\alpha.(\alpha \to \alpha)$ is needed for $(\lambda x.x)$, it will be the case that $(\lambda x.x)$ is embedded inside a larger term in a position where the abstraction will be λ^3-labelled. If it is desired to know what type will be assigned to the λ-term M in such a position, the type inference algorithm can be asked to type $(\mathbf{I}M)$ instead. The primary problem with this typing quirk is that the type derived for a free-standing λ-term does not indicate to the human viewer the actual possible types the λ-term can take in combination with other λ-terms. The type inference algorithm must pick some type, but, due to the lack of principal types, the type it picks can not be a most general type.

It may be desired to know the most general open type that can be assigned to a λ-term M, ignoring all of the possible rank-2 but not rank-1 final types. (This is different from **ML** typing in that rank-2 types are allowed in intermediate steps in the type derivation.) This is quite simple to do: simply ask the type inference algorithm the type of $(\mathbf{I}M)$. Any rank-1 type derivable for M in Λ_2^- is also derivable for $(\mathbf{I}M)$, but no rank-2 but not rank-1 types are derivable for $(\mathbf{I}M)$.

Similarly, it may also be desired to find a most general typing in which all types in the final sequent are open. To find such a typing for λ-term M with free variables $w_1, \ldots, w_n$, use the type inference algorithm to compute the type for the λ-term $(\mathbf{I}(\lambda w_1. \ldots .\lambda w_n.M))$, which will be of the shape $\rho_1 \to \cdots \to \rho_n \to \varphi$, where $\vec{\rho}, \varphi \in \mathbb{S}(0)$. In this case, any type-substitution instance of the following sequent will be derivable:

$$\{w_1 : \rho_1, \ldots, w_n : \rho_n\} \vdash M : \varphi$$

It may be desired to use specific closed types for some of the free or λ^2-bound variables of a λ-term, but to have the type inference algorithm compute most general open types for the rest of the free or λ^2-bound variables. Let the λ-term M have free variables $w_1, \ldots, w_n$ and let $act(M) = x_1 \ldots x_m$. It will be the case that

$$\theta\text{-nf}((M)^\lambda) \equiv (\lambda^2 x_1. \ldots .\lambda^2 x_m.N)$$

for some N which is not an abstraction and which contains no λ^2-bindings. Suppose we want to fix the type of x_1 as $\forall\alpha.(\alpha \to \alpha \to \alpha)$ but we wish the type inference algorithm to find most general open types for the rest of the free and λ^2-bound variables. To accomplish this, we can run the type inference algorithm on the λ-term

$$(\lambda^2 x_1.\mathbf{I}(\lambda^3 w_1. \ldots .\lambda^3 w_n.\lambda^3 x_2. \ldots .\lambda^3 x_m.N))$$

using the type assignment $A = \{x_1 : \forall\alpha.(\alpha \to \alpha \to \alpha)\}$, which will produce a type

$$A(x_1) \to \rho_1 \to \ldots \to \rho_n \to \psi_2 \to \ldots \to \psi_m \to \varphi$$

From this, we can conclude that any type-substitution instance (where $\forall$-bound variables are unchanged of course) of the following sequent is derivable:

$$A \cup \{w_1 : \rho_1, \ldots, w_n : \rho_n\} \vdash M : A(x_1) \to \psi_2 \to \ldots \to \psi_m \to \varphi$$

At times, we may want to assign more complex closed types to the free and λ^2-bound variables of a λ-term. It would be nice if the type inference algorithm would provide enough information so that we could know if a particular combination of closed types would work. Unfortunately, we do not currently have a method of knowing which closed types can be used without actually trying the type inference algorithm with that set of types assigned to the free and λ^2-bound variables.

References

Relevant documents not cited in the main text are [KMM90, Tiu90, Hen88].

[Dow93] G. Dowek. The undecidability of typability in the $\lambda\Pi$-calculus. In TLCA [TLCA93], pp. 139–145.

[Gir72] J.-Y. Girard. *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieur.* Thèse d'Etat, Université Paris VII, 1972.

[GMW79] M. J. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF*, vol. 78 of *LNCS*. Springer-Verlag, 1979.

[GRDR91] P. Giannini and S. Ronchi Della Rocca. Type inference in polymorphic type discipline. In *Theoretical Aspects Comput. Softw. : Int'l Conf.*, vol. 526 of *LNCS*, pp. 18–37. Springer-Verlag, Sept. 1991.

[Hen88] F. Henglein. Type inference and semi-unification. In *Proc. 1988 ACM Conf. LISP Funct. Program.*, Snowbird, Utah, U.S.A., July 25–27, 1988. ACM.

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

[KMM90] P. Kanellakis, H. Mairson, and J. C. Mitchell. Unification and ML type reconstruction. In *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, 1990.

[KT92] A. J. Kfoury and J. Tiuryn. Type reconstruction in finite-rank fragments of the second-order λ-calculus. *Inf. Comput.*, 98(2):228–257, June 1992.

[KTU90] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. In *15th Colloq. Trees Algebra Program.*, vol. 431 of *LNCS*, pp. 206–220. Springer-Verlag, 1990.

[KTU93] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Inf. Comput.*, 102(1):83–101, Jan. 1993.

[Lei83] D. Leivant. Polymorphic type inference. In *Conf. Rec. 10th Ann. ACM Symp. Princ. Program. Lang.*, pp. 88–98, Austin, TX, U.S.A., Jan. 24–26, 1983.

[Lei91] D. Leivant. Finitely stratified polymorphism. *Inf. Comput.*, 93(1):93–113, July 1991.

[McC84] N. McCracken. The typechecking of programs with implicit type structure. In *Semantics of Data Types : Int'l Symp.*, vol. 173 of *LNCS*, pp. 301–315, Sophia-Antipolis, France, June 1984. Springer-Verlag.

[Mil85] R. Milner. The standard ML core language. *Polymorphism*, 2(2), Oct. 1985.

[Pie92] B. Pierce. Bounded quantification is undecidable. In *Conf. Rec. 19th Ann. ACM Symp. Princ. Program. Lang.*, pp. 305–315. ACM, 1992.

[Rey74] J. C. Reynolds. Towards a theory of type structure. In *Symposium on Programming*, vol. 19 of *LNCS*, pp. 408–425, Paris, France, 1974. Springer-Verlag.

[Tiu90] J. Tiuryn. Type inference problems: a survey. In *Proc. Int'l Symp. Math. Found. Comput. Sci.*, vol. 452 of *LNCS*, pp. 105–120. Springer-Verlag, 1990.

[TLCA93] *Int'l Conf. Typed Lambda Calculi and Applications*, vol. 664 of *LNCS*. Springer-Verlag, Mar. 1993.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *IFIP Int'l Conf. Funct. Program. Comput. Arch.*, vol. 201 of *LNCS*. Springer-Verlag, 1985.

[Urz93] P. Urzyczyn. Type reconstruction in $\mathbf{F}_\omega$ is undecidable. In TLCA [TLCA93], pp. 418–432.

[Wel93] J. B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. Tech. Rep. 93-011, Comput. Sci. Dept., Boston Univ., 1993.