Open access • Proceedings Article • DOI:10.1145/62678.62697

Partial polymorphic type inference and higher-order unification — Source link

Frank Pfenning

Institutions: Carnegie Mellon University

Published on: 01 Jan 1988 - International Conference on Functional Programming

Topics: Type inference, Unification and Undecidable problem

Partial Polymorphic Type Inference and Higher-Order Unification

Frank Pfenning*
Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Abstract

We show that the problem of partial type inference in the nth-order polymorphic λ-calculus is equivalent to nth-order unification. On the one hand, this means that partial type inference in polymorphic λ-calculi of order 2 or higher is undecidable. On the other hand, higher-order unification is often tractable in practice, and our translation entails a very useful algorithm for partial type inference in the ω-order polymorphic λ-calculus. We present an implementation in λProlog in full.

1 Introduction

Even though polymorphism is generally regarded as a very desirable feature in typed programming languages, many languages (like ML) restrict themselves to an implicit polymorphism without explicit type abstraction or type application. This is partly due to the difficulty or even undecidability of type inference in languages with explicit polymorphism (see Boehm [2]). As a consequence, large amounts of type information must be given together with a program, making programming in such an explicitly typed language cumbersome.

This research was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract.

*The author can be reached via electronic mail on the ArpaNet as fp@cs.cmu.edu.

In this paper we examine the ω-order polymorphic λ-calculus as a core language that can serve as a basis for the implementation of syntactically much richer languages with explicit type abstraction and application. Many types and language constructs can be defined directly or introduced as typed constants. Definable types include products, sums, lists; constructs that can be included in the type checking merely with the addition of a new type constant include Mitchell and Plotkin's [19] **abstype** and **reps** for the definition and use of abstract data types.

In explicitly polymorphic languages, at least two different type inference problems arise which we call *full* and *partial* type inference. These different problems reflect different views of the underlying λ-calculus, either as type-assignments to untyped terms (the "Curry" view [5]) or as filling in information omitted from typed terms (the "Church" view [4]). The former leads to full type inference, the latter to partial type inference. More on this dichotomy in Section 2.1.

Full type inference means that arbitrary type abstractions and applications may be inserted into the untyped term in order to find a valid typing. Aspects of full type inference for the second-order polymorphic λ-calculus have been examined by Leivant [13], McCracken [14], and Mitchell [19], but the decidability question is still open.

We believe that full type inference, even though it may be decidable, is of limited use in programming practice. Types in a program provide more than just information for the compiler — they also provide succinct and formal documentation and thus help the programmer read, debug, and maintain his programs. If all type information is omitted, too many programs may have a correct typing without carrying the meaning intended by the programmer. The fact that there may be many different type assignments to a given term compounds this problem.

As a simple example consider $\lambda x\,.\,(x\,x)$ (the notation

is explained in Section 2). Clearly this term has a typing if we insert a type application, for example leading to $\lambda x{:}\Delta\theta\ .\ \theta \Rightarrow \theta\ .\ (x[\Delta\theta\ .\ \theta \Rightarrow \theta]x)$, which may or may not be what the programmer intended.

Partial polymorphic type inference, on the other hand, does not attempt to arbitrarily insert type abstractions and applications, but requires placeholders to be supplied by the user. Thus $\lambda x\ .\ (x\,x)$ would not type-check, but $\lambda x\ .\ (x[\,]x)$ would, since there is an explicit indication that x was intended to be a type abstraction. This is perhaps a little more verbose than the completely untyped term, but still much shorter than its fully typed counterpart above. Partial type inference has been investigated by Mitchell [21] and Boehm [2].

As the main result of the paper we show that partial polymorphic type inference in the nth-order polymorphic λ-calculus is equivalent to nth-order unification. This may be disheartening, since already second-order unification is undecidable as shown by Goldfarb in [9], and thus partial type inference is undecidable even for the second-order polymorphic λ-calculus (a slightly weaker version of this theorem was proven by Boehm in [2]).

However, we can make very good use of the converse, namely that partial polymorphic type inference can be solved through higher-order unification. Huet's complete algorithm for higher-order unification described in [11] has proven a very valuable tool with good computational properties when given realistic problems. It is used successfully in other areas like higher-order logic programming [16], natural deduction [22], automated theorem proving [1], and program derivation [12,24].

Somewhat unexpectedly, one can also recover the convenience of ML-style polymorphism by allowing the generic **let**. This extension (see Section 2.2) is convenient for the same reason it is convenient in ML, namely that partial type inference often does not completely determine the type of a term, but rather has free variables in its solution. The typing rule for the ML **let** constructs allows these variables to be instantiated differently at different occurrences of the term — in our extended setting we may do the same. Of course, the extension to the polymorphic λ-calculus means that the variables will sometimes be "higher-order", that is, they may be instantiated with arbitrary functions from types to types, etc.

Our implementation is written in Miller and Nadathur's λProlog language (see [16]). It is extremely well-suited for a problem of the kind described here, since higher-order unification is λProlog's inference engine. The program is short and concise (see Section 5). However, an implementation in support of a real programming language would need to be somewhat more sophisticated than our prototype. The search space should probably be explored through iterative deepening, rather than depth-first as in λProlog (which is incomplete). Also, it should sometimes return "maybe" when no valid typing can be found or contradiction discovered, and ask for more information from the programmer.

To date we have only run very preliminary experiments. However, when given realistic input, our prototype implementation performed well in discovering valid typings of expressions with relatively little explicit type information. Somewhat more surprisingly, it also showed good failure properties and most of the time detected when terms could not be typed. We have also identified cases where some simple, general improvements to the current λProlog interpreter would cause our program to return with failure where it currently follows an infinite computational branch.

2 The ω-Order Polymorphic λ-Calculus

In [8,7], Girard defines a powerful extension to Church's simply typed λ-calculus [3] and goes on to give a constructive proof of strong normalization for his system. A fragment of Girard's calculus was independently discovered by Reynolds [25] who introduced abstraction on type variables and application of functions to types in order to define explicitly polymorphic functions. Reynolds' calculus is known as the second-order polymorphic λ-calculus.

In this paper we consider the ω-order polymorphic λ-calculus which is an extension of Reynolds' system, but only a fragment of Girard's system since it omits existentially quantified types. Our presentation of the calculus consists contains three distinct syntactic categories: *kinds*, *types*, and *terms*. In Girard's presentation in [7], kinds are called *orders*.

Since our calculus is higher-order, in addition to types of terms, we have functions from types to types, etc. We will call every such object a *type*. The subset of those that is first-order, or, equivalently, of kind "Type", can actually be the type of a term. These and other properties of the calculus are summarized towards the end of this section.

Since we will need to be careful about the set of constants, the language is parameterized over a set of type constructors and a set of term constants. They are captured in the *kind signature* and *type signature*, respectively. We use K, M for kinds, $\alpha, \beta, \ldots$ for types, θ, ψ for type variables, σ for type constructors, $u, v, \ldots$ for terms, $x, y, \ldots$ for variables, and c for constants.

Definition 1 *The syntactic categories of* kind, type, *and* term *are defined through*

$$\begin{array}{lrcl} \textit{Kinds} & K & ::= & \text{Type} \mid K \to M \\ \textit{Types} & \alpha & ::= & \sigma \mid \theta \mid L\theta{:}K\,.\,\alpha \mid \alpha\,\beta \\ \textit{Terms} & u & ::= & c \mid x \mid \lambda x{:}\alpha\,.\,u \mid u\,v \mid \\ & & & \quad \Lambda\theta{:}K\,.\,u \mid u\,[\alpha] \end{array}$$

Definition 2 *A* kind signature Σ *is a set of pairs uniquely associating a kind with each type constructor. We define an extension* Σ^+ *of a given kind signature to include the constants* $\Rightarrow$:Type $\to$ Type $\to$ Type $\in \Sigma^+$ *and, for every valid kind* K, *also* Π_K:$(K \to \text{Type}) \to \text{Type} \in \Sigma^+$. *A* type signature C *is a set of pairs uniquely associating a type with each constant. A* context Γ *uniquely assigns kinds to type variables and types to term variables.*

Following common convention, we will sometimes omit an empty signature or context. In order not to conflict with the "," separating function arguments, we sometimes use "$\oplus$" to adjoin a pair to a signature or context.

The type constants in $\Rightarrow$ and Π_K play a special role: $\Rightarrow$ is the function type constructor, Π_K constructs types of polymorphic functions. The idea behind the type constructor Π_K is due to Church [4] who used Π as a constant to serve as a universal quantifier. If we restricted ourselves to the second order polymorphic calculus, we could express the type $\Delta\theta\,.\,\theta \Rightarrow \theta$ as $\Pi\,(L\theta\,.\,\theta \Rightarrow \theta)$. In the ω-order polymorphic λ-calculus we have to be a bit more verbose, since Δ may abstract over types of arbitrary kind. Thus, instead of one constant Π, we have a family of constants Π_K parameterized over the kind of the abstracted variable. Note that in Church's formulation of the simple theory of types, a constant Π also exists at every type.

In the second-order fragment of the polymorphic λ-calculus as defined above, one can explicitly define common data types and operations on them, such as natural numbers ($\text{int} \equiv \Delta\theta\,.\,(\theta \Rightarrow \theta) \Rightarrow (\theta \Rightarrow \theta)$), products, disjoint union, or lists ($\text{list} \equiv L\alpha\,.\,\Delta\theta\,.\,(\alpha \Rightarrow \theta \Rightarrow \theta) \Rightarrow \theta \Rightarrow \theta$).

The benefits of transcending the second-order polymorphic λ-calculus become visible in the example of abstract data types. Instead of new primitive constructs and type deduction rules, we can just introduce a new family of constants into the type signature and signature. This new type constructor is sigma_K (not to be confused with the symbol Σ for the type signature) and constants $\texttt{abs}_K$ and $\texttt{rep}_K$.

Example 3 (Abstract data types).

$$\begin{array}{rcl} \text{sigma}_K & \in & (K \to \text{Type}) \to \text{Type} \\ \texttt{abs}_K & \in & \Delta\alpha{:}K \to \text{Type}\ \Delta\beta{:}\text{Type}\,.\,\text{sigma}_K\,\alpha \Rightarrow \\ & & \quad (\Delta\theta{:}K\,.\,\alpha\,\theta \Rightarrow \beta) \Rightarrow \beta \\ \texttt{rep}_K & \in & \Delta\alpha{:}K \to \text{Type}\,.\,\Delta\theta{:}K\,.\,\alpha\,\theta \Rightarrow \text{sigma}_K\alpha \end{array}$$

In the formulation of the system with existential types through inference rules, a restriction is placed on β *in the definition of* $\texttt{abs}_K$, *namely that it not contain* θ. *Here that is unnecessary, since an attempt to substitute a type for* β *that contains* θ *free would lead to a renaming of* θ *in order to prevent a type clash. Thus this restriction is embedded into our system directly. For a discussion on how these constants can be used to represent abstract data types, transliterate work from Mitchell and Plotkin [20].*

In this shortened presentation we will not explicitly state the inference rules used to establish the validity of contexts or signatures. They hold no surprises.

We will regard α-convertible types and terms (with L and λ the respective binders) to be equal. Thus we will mostly ignore issues of variable renaming and name clashes.

In the inference rules of the polymorphic λ-calculus, we will allow conversions between $\beta\eta$-equivalent types. We define β and η conversions of types as is done usually on terms. A β-redex then takes the form $(L\theta{:}K\,.\,\alpha)\,\gamma$.

Next we define the *judgments* of the inference system that allows us to find valid types for terms and kinds for types.

Definition 4 *A* judgment *is one of the following assertions:*

$\vdash \Sigma$ kindsig	Σ *is a valid kind signature*
$\vdash_\Sigma C$ typesig	C *is a valid type signature*
$\vdash_{\Sigma,c} \Gamma$ context	Γ *is a valid context*
$\vdash K \in$ kind	K *is a valid kind*
$\Gamma \vdash_\Sigma \alpha \in K$	α *has kind* K
$\Gamma \vdash_{\Sigma,c} u \in \alpha$	u *has type* α

We now define the inference system defining valid judgments. It is divided into groups corresponding to the different judgments which in turn correspond to the different syntactic categories. Every *kind* that is syntactically well-formed is also valid, so we omit the rules for valid kinds.

Definition 5 (Valid Type).

$$\frac{\vdash_\Sigma \Gamma \text{ context} \qquad \sigma{:}K \in \Sigma^+}{\Gamma \vdash_\Sigma \sigma \in K}$$

$$\frac{\vdash_\Sigma \Gamma \text{ context} \qquad \theta{:}K \in \Gamma}{\Gamma \vdash_\Sigma \theta \in K}$$

$$\frac{\Gamma \vdash_\Sigma K \in \text{kind} \qquad \Gamma, \theta{:}K \vdash_\Sigma \alpha \in M}{\Gamma \vdash_\Sigma L\theta{:}K \,.\, \alpha \in K \to M}$$

$$\frac{\Gamma \vdash_\Sigma \alpha \in K \to M \qquad \Gamma \vdash_\Sigma \beta \in K}{\Gamma \vdash_\Sigma \alpha\,\beta \in M}$$

Definition 6 (Valid Term).

$$\frac{\vdash_\Sigma \Gamma \text{ context} \qquad c{:}\alpha \in C}{\Gamma \vdash_{\Sigma,c} c \in \alpha}$$

$$\frac{\vdash_\Sigma \Gamma \text{ context} \qquad x{:}\alpha \in \Gamma}{\Gamma \vdash_{\Sigma,c} x \in \alpha}$$

$$\frac{\Gamma, x{:}\alpha \vdash_{\Sigma,c} u \in \beta \qquad \Gamma \vdash_{\Sigma,c} \alpha \in \text{Type}}{\Gamma \vdash_{\Sigma,c} \lambda x{:}\alpha \,.\, u \in \alpha \Rightarrow \beta}$$

$$\frac{\Gamma \vdash_{\Sigma,c} u \in \alpha \Rightarrow \beta \qquad \Gamma \vdash_{\Sigma,c} v \in \alpha}{\Gamma \vdash_{\Sigma,c} u\,v \in \beta}$$

$$\frac{\Gamma, \theta{:}K \vdash_{\Sigma,c} u \in \beta\theta \qquad \vdash K \in \text{kind}}{\Gamma \vdash_{\Sigma,c} \Lambda\theta{:}K \,.\, u \in \Pi_K \beta}$$

$$\frac{\Gamma \vdash_{\Sigma,c} u \in \Pi_K \beta \qquad \Gamma \vdash_{\Sigma,c} \alpha \in K}{\Gamma \vdash_{\Sigma,c} u\,[\alpha] \in \beta\,\alpha}$$

$$\frac{\Gamma \vdash_{\Sigma,c} u \in \alpha \qquad \alpha \approx \beta \qquad \Gamma \vdash_\Sigma \beta \in \text{Type}}{\Gamma \vdash_{\Sigma,c} u \in \beta}$$

Some of the antecedents of the rules are redundant but included for technical reasons. In particular, in the rule for λ the antecedent asserting that α is a valid type is unnecessary, since it follows from the proof that $\Gamma, x{:}\alpha$ is a valid environment. Similarly, we do not need the fact that K is a valid kind in the rule for Λ.

During the remainder of the paper, we will make use of some fundamental properties of the calculus. There are other theorems of interest, like strong normalization for terms, which we will not need.

Lemma 7 *1. If* $\Gamma \vdash_\Sigma \alpha \in K$ *then* $\vdash \Sigma$ kindsig *and* $\vdash_\Sigma \Gamma$ context *and* $\vdash K \in$ kind.

2. If $\Gamma \vdash_{\Sigma,c} u \in \alpha$ *then* $\vdash \Sigma$ kindsig *and* $\vdash_\Sigma \Gamma$ context *and* $\Gamma \vdash_\Sigma \alpha \in$ Type.

Lemma 8 (Interpretation of types). *The types of the polymorphic λ-calculus form a simply typed λ-calculus. We interpret kinds as types and types as terms. Under this interpretation we obtain a single base type* Type *with a binary function constant* $\Rightarrow$:Type $\to$ Type $\to$ Type *and a constant* Π_K:($K \to$ Type) $\to$ Type *for each type K. More term constants may exist depending on the kind signature Σ.*

Theorem 9 (Normal forms for types). *If* $\Gamma \vdash_\Sigma \alpha \in K$ *then α has a unique long $\beta\eta$-normal form.*

2.1 Partial Type Inference

In this section we define the problem of partial type inference. This is distinct from the problem of *full type inference* which is to decide whether, given a completely untyped term in the λ-calculus, there is a way of inserting types of bound variables, type abstractions, and type applications such that the resulting term in the polymorphic λ-calculus is well-typed. Even though the problems are closely related, undecidability of partial type inference does not imply undecidability of full type inference.

Partial *vs.* full type inference reflects two views of the underlying λ-calculus. This difference in approach has sometimes been described as the "Church *vs.* Curry" debate. Our formulation of the ω-order λ-calculus is a natural extension of Church's definition of the simply typed λ-calculus [4], in which types are embedded in terms. In Curry's formulation [5] (see also Hindley [10]), types may be considered as properties of terms in the untyped λ-calculus.

Curry's view leads to the problem of full type inference which has been discussed in the literature in various places (see, for example, Leivant [13], McCracken [14], and Mitchell [19]), but whose decidability is still open.

Church's view leads to the problem of partial type inference some aspects of which have been discussed by Mitchell in [21] and Boehm in [2], and which is the subject of this paper.

We adopted the name "partial type inference" from Boehm, but it is somewhat misleading. We would like to emphasize that it is no more or no less complete than the full type inference problem, but simply defined for a different view of a calculus of second- and higher-order types. In our opinion both views and associated type inference problems are theoretically interesting — which one has greater practical relevance remains to be seen.

This difference may be somewhat difficult to appreciate, since, in calculi with only simple types, the two views of type inference coincide, or rather, essentially the same type inference algorithm (based on first-order unification) solves both problems (see, for example, [17]).

Definition 10 *The definition of a* partially typed term *can be obtained from the definition of a* term *by adding two alternatives,*

$$\bar{u} ::= \ldots \mid \lambda x \,.\, \bar{u} \mid \bar{u}\,[\,]_K$$

where $\bar{u}$ stands for a partially typed term.

In the special case of the second-order polymorphic λ-calculus, we omit the subscript to the "[]" since it must be "Type".

We now need a new judgment expressing that a partially typed term $\bar{u}$ is the result of erasing some type information from a term u. We will again define this relation through an inference system, since we need to maintain a context.

Definition 11 *Formally we define the judgment $\Gamma \vdash_{\Sigma,c} \bar{u} \leq u$ (read: $\bar{u}$ has less type information than u) through an following inference system updating Γ. From the following informal definition it should be easy to see how to construct this inference system.*

$$\begin{array}{rcll} c & \leq & c & \text{for } c \in \Sigma^+ \\ x & \leq & x & \\ \lambda x{:}\alpha \,.\, \bar{u} & \leq & \lambda x{:}\alpha \,.\, u & \text{if } \bar{u} \leq u \\ \lambda x \,.\, \bar{u} & \leq & \lambda x{:}\alpha \,.\, u & \text{if } \bar{u} \leq u \\ \bar{u}\,\bar{v} & \leq & u\,v & \text{if } \bar{u} \leq u \text{ and } \bar{v} \leq v \\ \Lambda\theta{:}K \,.\, \bar{u} & \leq & \Lambda\theta{:}K \,.\, u & \text{if } \bar{u} \leq u \\ \bar{u}\,[\alpha] & \leq & u\,[\alpha] & \text{if } \bar{u} \leq u \\ \bar{u}\,[\,]_K & \leq & u\,[\alpha] & \text{if } \bar{u} \leq u \text{ and } \alpha \in K \end{array}$$

The crucial difference to the full type inference problem lies in the fact that we do not have $\bar{u} \leq u[\alpha]$ or $\bar{u} \leq \Lambda\theta{:}K \,.\, u$ for $\bar{u} \leq u$.

Definition 12 (Partial type inference). *A term u is a* fully typed instance *of a partially typed term $\bar{u}$ iff $\vdash_{\Sigma,c} \bar{u} \leq u$. The problem of* partial type inference *is to find a fully typed instance of a given partially typed term.*

Unlike in the case of the simply-typed λ calculus, no single most general or *principal* type-schemas exist in the polymorphic calculus. Consider the example from the introduction, $\lambda x \,.\, (x\,[\,]\,x)$. There are two most general solutions to this type inference problem, namely

$$\lambda x{:}(\Delta\theta \,.\, \theta \Rightarrow \delta\,\theta) \,.\, (x\,[\Delta\theta \,.\, \theta \Rightarrow \delta\,\theta]\,x)$$
$$\lambda x{:}(\Delta\theta \,.\, \theta) \,.\, (x\,[(\Delta\theta \,.\, \theta) \Rightarrow \gamma]\,x)$$

where δ:Type $\rightarrow$ Type and γ:Type are free variables. Any instance of these typings will be a solution, and any solution will be an instance of the ones given above.

2.2 Adding Generic Polymorphism

Type inference in the programming language ML [18] differs from type inference for the simply typed λ-calculus only in one aspect: it uses *generic type variables* for variables bound by **let**. Thus, for example,

$$\textbf{let}\, f = \lambda x \,.\, x \,\textbf{in}\, (f\,1, f\,\mathsf{true})$$

is type-correct, since $\lambda x \,.\, x$ has principle type $\alpha \Rightarrow \alpha$ for a type variable α and this type variable may be instantiated differently at different occurrences of f in the scope of the binding on f (and is thus called *generic*). Hence **let** cannot be treated merely as syntactic sugar, since the expanded version of the example above,

$$(\lambda f \,.\, (f\,1, f\,\mathsf{true}))\,(\lambda x \,.\, x)$$

would not be type-correct.

Explicit polymorphism does eliminate the formal need for **let**, since the programmer can always insert explicit type abstractions and applications to achieve the effect of generic type variables. Thus the example above could be written as

$$(\lambda f \,.\, (f\,[\,]\,1, f\,[\,]\,\mathsf{true}))\,(\Lambda\theta\ \lambda x{:}\theta \,.\, x).$$

From the practical point of view, however, the generic typing possible through the **let** construct has enormous value.

Surprisingly, generic typing and partial polymorphic type inference seem to be orthogonal issues, in the sense that generic typing may be added to the polymorphic λ-calculus in a consistent and practically useful manner. We could for example define:

$$\begin{array}{rcl} \textbf{let}\ \mathtt{cons}^* & = & \Lambda\alpha\ \lambda x{:}\alpha\ \lambda y{:}(\mathsf{list}\,\alpha) \,.\, \Lambda\theta \\ & & \quad \lambda f{:}\alpha \Rightarrow \theta \Rightarrow \theta\ \lambda z{:}\theta \,.\, f\,x\,(y\,[\alpha]\,f\,z) \\ \textbf{let}\ \mathtt{cons} & = & \mathtt{cons}^*\,[\,] \end{array}$$

Now one can freely use "cons" as one is used to in ML as in "cons 1 l". The generic typing of cons achieves the effect (during type inference) of substituting cons* []

where cons appears, and therefore cons 1 l will typecheck.

In this manner all the convenience of ML-style polymorphism can be recovered without giving up the power of being able to explicitly abstract over types when desired. The implementation of type inference in the presence of **let** that we give in the Appendix is somewhat different from that in ML. This is due to the fact that principal type-schemas no longer exist, and that we therefore cannot simply record the principal type of a variable in an environment. Instead we carry out the substitution as indicated above, in effect implementing the inference rule

$$\frac{\Gamma \vdash_{\Sigma,c} u \in \beta \qquad \Gamma \vdash_{\Sigma,c} (u/x)\, v \in \alpha}{\Gamma \vdash_{\Sigma,c} \textbf{let } x = u \textbf{ in } v \;\in\; \alpha}$$

3 Partial Type Inference through Higher-Order Unification

In this section we show how the partial type inference problem can be solved using higher-order unification. More precisely, the partial type inference problem in the nth-order polymorphic λ-calculus can be reduced to nth-order unification (see Section 3.3). Although higher-order unification is in general undecidable, Huet's [11] algorithm is practical in many cases. So here, too: the sort of unification problem that is extracted from a partial type inference problem is very tractable in practice.

3.1 A Formulation of Higher-Order Unification

Our formulation of unification will be somewhat unusual in that we present it as a deductive system. The conventional presentation of unification can be seen as a special case of our formulation in which formulas begin with a sequence of existential quantifiers followed by only conjunctions.

There are methods for reducing this more general problem to the conventional unification problem. One can either use Skolemization (without increase in order), or lifting. For a discussion of these methods see Paulson [22,23] and Miller [15]. An extension of Huet's algorithm for higher-order unification that tries to find a proof of F directly is a subject of current research.

Definition 13 (Formulas of $\mathcal{U}$). *The set of formulas of the logical system $\mathcal{U}$ is defined recursively through*

$$F ::= \alpha \doteq \beta \mid F \wedge G \mid \exists\psi{:}K\,.\,F \mid \forall\theta{:}K\,.\,F$$

where $F, G, \ldots$ stand for formulas. $(\alpha/\psi)F$ is our notation for substitution of α for free occurrences of ψ in F, possibly renaming some bound variables.

Definition 14 (Inference rules of $\mathcal{U}$). *The only new judgment in the logic $\mathcal{U}$ is $\Theta \Vdash_\Sigma F$. It is defined through the following inference rules.*

$$\frac{\Theta \vdash_\Sigma \alpha \in \text{Type} \qquad \alpha \approx \beta \qquad \Theta \vdash_\Sigma \beta \in \text{Type}}{\Theta \Vdash_\Sigma \alpha = \beta}$$

$$\frac{\Theta \Vdash_\Sigma F \qquad \Theta \Vdash_\Sigma G}{\Theta \Vdash_\Sigma F \wedge G}$$

$$\frac{\Theta \Vdash_\Sigma (\alpha/\psi)F \qquad \Theta \vdash_\Sigma \alpha \in K}{\Theta \Vdash_\Sigma \exists\psi{:}K\,.\,F}$$

$$\frac{\Theta, \theta{:}K \Vdash F}{\Theta \Vdash_\Sigma \forall\theta{:}K\,.\,F}$$

3.2 Interpreting Type Inference as Unification

We now define a translation from the type inference problem to the theorem proving problem in $\mathcal{U}$. The motivation behind this translation is Theorem 16.

Definition 15 (Translation from partial type inference to theorem proving in $\mathcal{U}$). *Given a context Γ, a partially typed term $\bar{u}$, and a type α such that $\Gamma \vdash_\Sigma \alpha \in$ Type, we define $V(\Gamma, \bar{u}, \alpha)$ which constructs a formula F in $\mathcal{U}$. The definition is by induction and is shown in Figure 1.*

Theorem 16 *Given a partially typed term $\bar{u}$. Then the following are equivalent.*

1. $\Vdash \exists\psi{:}\text{Type}\,.\,V(\langle\rangle, \bar{u}, \psi)$.
2. *There is a (fully typed) term u such that $\bar{u} \leq u$ and a type β such that $\vdash_\Sigma \beta \in$ Type and $\vdash_{\Sigma,c} u \in \beta$.*

Moreover, this correspondence is constructive, that is, every deduction of 1 in $\mathcal{U}$ yields a u satisfying 2 and vice versa.

Proof: From Lemmas 17 and 18. □

In the formulation of the crucial lemmas, we use the notation Θ_Γ for result of erasing variable/type pairs from Γ, leaving only type variable/kind pairs.

$$
\begin{aligned}
V(\Gamma, c, \alpha) &= \alpha \doteq \beta \quad \textit{for} \quad c{:}\beta \in \mathcal{C} \\
V(\Gamma, x, \alpha) &= \alpha \doteq \beta \quad \textit{for} \quad x{:}\beta \in \Gamma \\
V(\Gamma, \lambda x{:}\beta \,.\, \bar{u}, \alpha) &= \exists \psi{:}\mathrm{Type} \,.\, \alpha \doteq (\beta \Rightarrow \psi) \wedge V(\Gamma \oplus x{:}\beta, \bar{u}, \psi) \\
V(\Gamma, \lambda x \,.\, \bar{u}, \alpha) &= \exists \beta{:}\mathrm{Type}\, \exists \psi{:}\mathrm{Type} \,.\, \alpha \doteq (\beta \Rightarrow \psi) \wedge V(\Gamma \oplus x{:}\beta, \bar{u}, \psi) \\
V(\Gamma, \bar{u}\,\bar{v}, \alpha) &= \exists \psi{:}\mathrm{Type} \,.\, V(\Gamma, \bar{u}, \psi \Rightarrow \alpha) \wedge V(\Gamma, \bar{v}, \psi) \\
V(\Gamma, \Lambda\theta{:}K \,.\, \bar{u}, \alpha) &= \exists \psi{:}K \rightarrow \mathrm{Type} \,.\, \alpha \doteq \Pi_K \psi \wedge \forall \theta{:}K \,.\, V(\Gamma \oplus \theta{:}K, \bar{u}, \psi\,\theta) \\
V(\Gamma, \bar{u}\,[\beta], \alpha) &= \exists \psi{:}K \rightarrow \mathrm{Type} \,.\, \alpha \doteq \psi\,\beta \wedge V(\Gamma, \bar{u}, \Pi_K \psi) \quad \textit{where} \quad \Gamma \vdash_\Sigma \beta \in K \\
V(\Gamma, \bar{u}\,[\,]_K, \alpha) &= \exists \beta{:}K\, \exists \psi{:}K \rightarrow \mathrm{Type} \,.\, \alpha \doteq \psi\,\beta \wedge V(\Gamma, \bar{u}, \Pi_K \psi)
\end{aligned}
$$

Figure 1: Definition of translation V.

Lemma 17 *Given a valid context Γ, a partially typed term $\bar{u}$, a type α such that $\vdash_\Sigma \alpha \in \mathrm{Type}$, and $\Theta_\Gamma \Vdash_\Sigma V(\Gamma, \bar{u}, \alpha)$. Then there is a (fully typed) term $u \geq \bar{u}$ such that $\Gamma \vdash_{\Sigma,c} u \in \alpha$.*

Proof: By induction on the deduction of $\Theta_\Gamma \Vdash_\Sigma V(\Gamma, \bar{u}, \alpha)$.

□

Lemma 18 *Given a partially typed term $\bar{u}$. Assume there is a (fully typed) term u with $\bar{u} \leq u$ and a type α such that $\Gamma \vdash_{\Sigma,c} u \in \alpha$. Then for every partially typed $\bar{u}^* \leq u$ and for every $\alpha^* \approx \alpha$ such that $\Gamma \vdash_\Sigma \alpha^* \in \mathrm{Type}$, we have $\Theta_\Gamma \Vdash_\Sigma V(\Gamma, \bar{u}^*, \alpha^*)$.*

Proof: By induction on the deduction of $\Gamma \vdash_{\Sigma,c} u \in \alpha$.

□

We have thus established that the partial type inference problem can be reduced to the problem of proving a theorem in the system $\mathcal{U}$. By an earlier remark, this also means a reduction to the conventional notion of higher-order unification. In particular, one can apply Huet's algorithm for higher-order unification to the type inference problem.

We can allow or disallow the "absurd" type $\Delta\theta{:}\mathrm{Type} \,.\, \theta$ as a result of the unification problem. This is very easily implemented by rejecting solutions containing this type.

3.3 Order of Partial Type Inference and Unification

We have not yet exploited all the information in the definition of the function V that translates a partial type inference problem into a higher-order unification problem. The correspondence can be made more precise: the type inference problem for the nth-order polymorphic λ-calculus can be solved by nth-order unification. The converse is also true (see Theorem 23).

We lack the space to formally define the order o of the type inference problem, but it is a straightforward generalization of the usual notion of order. Type inference for the simply typed λ-calculus is first-order, type inference for the second-order polymorphic λ-calculus is second-order, and higher types depend on the occurrence of higher-order functions between types. Note that constants (like Π_K) of order $n+1$ may appear in terms of order n. The order of the higher-order unification is the usual notion of order for the simply typed lambda calculus.

Theorem 19 *Given a partially typed term $\bar{u}$ of order $o(\bar{u}) = n$. Then the associated unification problem has order n, that is, $o(\exists \psi{:}\mathrm{Type} \,.\, V(\langle\rangle, \bar{u}, \psi)) = n$.*

Proof: By induction on $\bar{u}$.

□

4 Higher-Order Unifiability through Partial Type Inference

In this section we show that there is a translation from the problem of higher-order unifiability into partial type inference in the ω-order polymorphic λ-calculus. More concretely, given a higher-order unification problem, that is a formula F in $\mathcal{U}$, we will construct a partially typed term $\bar{u}$ such that $\bar{u}$ has a valid typing iff F is a theorem. For the second-order polymorphic λ-calculus (with some restrictions), a similar result was proven by Boehm in [2]. Here we generalize the result to arbitrary order and also remove the restriction that required identifiers of type $\Delta\theta \,.\, \theta$ (or other constants in the language).

This result establishes the undecidability of the type inference problem for every polymorphic λ-calculus of order 2 or more. Some care must be taken to determine the exact conditions under which the translation is order-preserving and does not result in a type inference problem of higher order (see Corollary 24).

We now present the translation. Later we will discuss the circumstances under which this translation is order-preserving.

Definition 20 (Type closure). *Given a type* $\alpha \in K_1 \to \cdots \to K_n \to \text{Type}$. *Then* $\bar{\alpha}$, *the* closure *of* α, *is the type* $\Delta\theta_1{:}K_1 \ldots \Delta\theta_n{:}K_n \,.\, \alpha\,\theta_1 \ldots \theta_n$.

Note that the closure of *any* (higher-order) type α will by such that $\bar{\alpha} \in \text{Type}$.

We now define the translation from formulas F in $\mathcal{U}$ to partially typed terms $\bar{u}$ such that F is provable iff the corresponding term $\bar{u}$ has a valid type (see Theorem 23).

For the sake of brevity, we restrict the definition of B to the case of the second-order polymorphic λ-calculus. Through the use of type closures B can easily be extended to the full ω-order calculus. B will be an auxiliary function in the translation D (see the following definition). Intuitively, a valid typing of $B(\langle\rangle, \langle\rangle, x, \alpha)$ will force x to have type α.

Definition 21 *We define* $B(\Psi, \Gamma, x, \alpha)$ *by induction on the structure of* α, *where* α *is in long* $\beta\eta$*-normal form. It is given in Figure 2.*

The definition of the translation D is given for the full calculus, since the difference between the second- and higher-order versions is minimal (in the second-order case it is not necessary to take the type closure of ψ in the clause for an existentially quantified formula).

Definition 22 *We define the translation* D *by induction on the structure of a formula* F *in* $\mathcal{U}$. *It is given in Figure 3.*

In D, Ψ keeps track of free type variables in the equations and their kinds, and Γ keeps track of the universally quantified type variables.

Note that we could not have simply let $D(\Psi, \Gamma, \alpha \doteq \beta) = \lambda x{:}\alpha\ \lambda y{:}\beta \,.\, \lambda f \,.\, f\,x\,(f\,y\,(\lambda z \,.\, z))$, since α and β contain free variables from the removal of existentially quantified variables. Therefore the right-hand side is not a partially typed term. Thus the purpose of B is to analyze the structure of α and β and create a partially typed term (without any free type variables) that can be typed iff there is a type substitution for the free type variables in α and β that would make $\alpha \doteq \beta$ provable.

Theorem 23 *Given a formula* F *of the system* $\mathcal{U}$. *Then* $\Vdash_\Sigma F$ *iff there is a fully typed term* $u \geq D(\langle\rangle, \langle\rangle, F)$ *and a type* β *such that* $\vdash_{\Sigma,c} u \in \beta$.

Proof: In each direction one generalizes by quantifying over contexts and then proves the translation correct by induction on the structure of the deduction.

□

Corollary 24 *Partial type inference for the second-order polymorphic* λ*-calculus is undecidable.*

Proof: By inspecting the rules one can see that only translation of $\forall\theta{:}K \,.\, F$ leads to a partial type inference problem of higher-order than the unification problem. However, the second-order unifiability problem is undecidable without the presence of the $\forall$ quantifier (a conventional formulation of a unification problem is equivalent to a formula F with an existential quantifier prefix and a matrix consisting merely of conjunctions). "$\Rightarrow$:Type $\to$ Type $\to$ Type" is present in Σ^+ and serves as the one binary function constant required for undecidability of second-order unification (see Goldfarb [9]). This observation is important, since the purely monadic second-order unification problem is indeed decidable (see Farmer's algorithm in [6])

□

Note that the type inference problems generated by the translation D contain very little explicit type information. In fact, we only need type variables that are explicitly abstracted over (with Λ) somewhere in the term. The only exception is the case of a type constructor σ. This exception can be eliminated if the language satisfies a simple and natural closure condition (roughly, that every type constructor in the kind signature has a corresponding term constructor).

5 The λProlog Implementation

The full code of our implementation in λProlog is provided in an Appendix on the following page. Unfortunately, space does not permit to explain the language and our implementation of the algorithm in detail. The implementation is concise, since higher-order unification, including the correct treatment of universally quantified variables, is the main computational device of λProlog.

```
% A formulation of the simply typed lambda calculus in which
% types may be omitted.

module tp_types.

infix 110 ==>   xfy.    % right associative: function types.
infix 120 @     yfx.    % left associative: application.

kind  tp        type.   % types.
kind  term      type.   % terms.

type  ==>       tp -> tp -> tp.   % function type constructor.

type  lam       tp -> (term -> term) -> term.  % lambda.
type  olam      (term -> term) -> term.        % type omitted.
type  @         term -> term -> term.          % application.

-------------------------------------------------------------------
% Type inference for the simply typed lambda calculus.

module tp_simple.

import tp_types.

type  termtype  term -> term -> tp -> o.
type  vartype   term -> tp -> o.
type  contype   term -> tp -> o.                % external.

% termtype U V Alpha
% takes a partially typed term U and returns fully typed term V.
% Alpha may be a constraint (the type we know that U should
% have) or a result (the infered type of U).

% vartype X Beta
% records the types that were infered for bound variables.
% The types of free variables should be given in the environment,
% but may of course be variable.  See tp_simple_ex for an example.

% The simplest clause here would be
% termtype (olam     U) (lam Beta V) (Beta ==> Gamma) :-
%    (pi X\ (vartype X Beta => termtype (U X) (V X) Gamma)).
% but that would be unnecessarily inefficient.

termtype (olam     U) (lam Beta V) Alpha :- !,
  Alpha = (Beta ==> Gamma),
  (pi X\ (vartype X Beta => termtype (U X) (V X) Gamma)).

termtype (lam Beta U) (lam Beta V) Alpha :- !,
  Alpha = (Beta ==> Gamma),
  (pi X\ (vartype X Beta => termtype (U X) (V X) Gamma)).

termtype (U0 @ U1) (V0 @ V1) Alpha :- !,
  termtype U0 V0 (Beta ==> Alpha),
  termtype U1 V1 Beta.

termtype X X Alpha :- vartype X Alpha.

termtype C C Alpha :- contype C Alpha.
```

```
-------------------------------------------------------------------
% The language extension for the omega-order polymorphic lambda calculus

module tp_polytypes.

infix   120 &   yfx.            % left assoc: type application.
postfix 120 &&  yf.             % omitted type application.

type  dd        (A -> tp) -> tp.              % universal type constructor.

type  ll        (A -> term) -> term.          % type abstraction.
type  &         term -> tp -> term.           % type application.
type  &&        term -> term.                 % type omitted.

-------------------------------------------------------------------
% Type inference for the omega-order polymorphic lambda calculus.

module tp_poly.

import tp_simple tp_polytypes tp_types.

termtype (ll U) (ll V) Alpha :- !,
   Alpha = (dd Beta),
   (pi Theta\ (termtype (U Theta) (V Theta) (Beta Theta))).

termtype (U &&    ) (V & Gamma) Alpha :- !,
  Alpha = (Beta Gamma),
  termtype U V (dd Beta).

termtype (U & Gamma) (V & Gamma) Alpha :- !,
  Alpha = (Beta Gamma),
  termtype U V (dd Beta).

-------------------------------------------------------------------
% Extend language and type inference to generically type 'let'.
% May be included with tp_simple or tp_poly.

module tp_ml.

import tp_types.

type  termtype  term -> term -> tp -> o.
type  vartype   term -> tp -> o.

type  tlet  (term -> term) -> term -> term.   % generically typed let.
type  gvarbinding  term -> term -> o.

% gvarbinding X V0
% remembers that X is a generically typed variable with "definition"
% V0.  V0 and therefore its free type variables are replicated wherever
% X appears.

termtype (tlet U0 V0) (tlet U1 V1) Alpha :- !,
  termtype V0 V1 ArgType,
  (pi X\ (gvarbinding X V0 => termtype (U0 X) (U1 X) Alpha)).

vartype X Alpha :- gvarbinding X V0, !, termtype V0 V1 Alpha.
```

$$
\begin{aligned}
B(\Psi,\Gamma,x,\theta) &= \lambda y{:}\theta\,.\,\lambda f\,.\,f\,x\,(f\,y\,(\lambda g\,.\,g)) \qquad \textit{where}\quad \theta{:}\mathrm{Type}\in\Gamma\\
B(\Psi,\Gamma,x,\psi) &= \lambda f\,.\,f\,x\,(f\,y\,(\lambda g\,.\,g)) \qquad \textit{where}\quad y{:}\psi\in\Psi\\
B(\Psi,\Gamma,x,\sigma\,\alpha_1\ldots\alpha_n) &= \lambda x_1\ldots\lambda x_n\,.\,\lambda z{:}\Delta\theta_1\ldots\Delta\theta_n\,.\,\theta_1\Rightarrow\cdots\Rightarrow\theta_n\Rightarrow\sigma\,\theta_1\ldots\theta_n\,.\\
&\qquad \lambda f\,.\,f\,x\,(f\,(z\,[\,]\ldots[\,]\,x_1\ldots x_n)\,(\lambda g\,.\,g\,B(\Psi,\Gamma,x_1,\alpha_1)\ldots B(\Psi,\Gamma,x_n,\alpha_n)))\\
B(\Psi,\Gamma,x,\alpha\Rightarrow\beta) &= \lambda y\,\lambda z\,.\,\lambda f\,.\,f\,(x\,y)\,(f\,z\,(\lambda g\,.\,g\,B(\Psi,\Gamma,y,\alpha)\,B(\Psi,\Gamma,z,\beta)))\\
B(\Psi,\Gamma,x,\Delta\theta\,.\,\alpha) &= \lambda y\,\Lambda\theta\,.\,\lambda f\,.\,f\,(x\,[\theta])\,(f\,y\,(\lambda g\,.\,g\,B(\Psi,\Gamma\oplus\theta{:}K,y,\alpha)))
\end{aligned}
$$

Figure 2: Definition of translation B.

$$
\begin{aligned}
D(\Psi,\Gamma,\alpha\doteq\beta) &= \lambda x\,\lambda y\,.\,\lambda f\,.\,f\,x\,(f\,y\,(\lambda g\,.\,g\,B(\Psi,\Gamma,x,\alpha)\,B(\Psi,\Gamma,y,\beta)))\\
D(\Psi,\Gamma,F\wedge G) &= \lambda g\,.\,g\;D(\Psi,\Gamma,F)\;D(\Psi,\Gamma,G)\\
D(\Psi,\Gamma,\exists\psi{:}K\,.\,F) &= \lambda x\,.\,D(\Psi\oplus\psi{:}K\oplus x{:}\bar{\psi},\Gamma,F)\\
D(\Psi,\Gamma,\forall\theta{:}K\,.\,F) &= \Lambda\theta{:}K\,.\,D(\Psi,\Gamma\oplus\theta{:}K,F)
\end{aligned}
$$

Figure 3: Definition of translation D.

6 References

[1] Peter B. Andrews, Dale Miller, Eve Cohen, and Frank Pfenning. Automating higher-order logic. *Contemporary Mathematics*, 29:169–192, August 1984.

[2] Hans-J. Boehm. Partial polymorphic type inference is undecidable. In *26th Annual Symposium on Foundations of Computer Science*, pages 339–345, IEEE, October 1985.

[3] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, New Jersey, 1941.

[4] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[5] H. B. Curry and R. Feys. *Combinatory Logic.* North-Holland, Amsterdam, 1958.

[6] William M. Farmer. *A Unification Algorithm for Second-Order Monadic Terms.* Technical Report, Mitre Corporation, Bedford, Massachusetts, June 1986.

[7] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordere supérieur.* PhD thesis, Université Paris VII, 1972.

[8] Jean-Yves Girard. Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92, North-Holland Publishing Co., Amsterdam, London, 1971.

[9] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[10] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[11] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[12] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[13] Daniel Leivant. Polymorphic type inference. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages*, ACM, 1983.

[14] Nancy McCracken. The typechecking of programs with implicit type structure. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data*

Types, pages 301–315, Springer-Verlag LNCS 173, 1984.

[15] Dale A. Miller. Unification under mixed prefixes. 1987. Unpublished manuscript.

[16] Dale A. Miller and Gopalan Nadathur. Higher-order logic programming. In *Proceedings of the Third International Conference on Logic Programming*, Springer Verlag, July 1986.

[17] Arthur J. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, August 1978.

[18] Robin Milner. The Standard ML core language. *Polymorphism*, II(2), October 1985.

[19] John Mitchell. Type inference and type containment. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 257–277, Springer-Verlag LNCS 173, 1984.

[20] John Mitchell and Gordon Plotkin. Abstract types have existential type. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 37–51, ACM, January 1985.

[21] John C. Mitchell. Second-order unification and types. June 1984. Unpublished notes.

[22] Lawrence Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[23] Lawrence C. Paulson. *The Representation of Logics in Higher-Order Logic*. Technical Report 113, University of Cambridge, Cambridge, England, August 1987.

[24] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation*, ACM, June 1988. To appear.

[25] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, Springer-Verlag LNCS 19, New York, 1974.