On the Undecidability of Partial Polymorphic Type Reconstruction

Frank Pfenning

January 1992

CMU-CS-92-105

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We prove that partial type reconstruction for the pure polymorphic λ-calculus is undecidable by a reduction from the second-order unification problem, extending a previous result by H.-J. Boehm. We show further that partial type reconstruction remains undecidable even in a very small predicative fragment of the polymorphic λ-calculus, which implies undecidability of partial type reconstruction for λ^{ML} as introduced by Harper, Mitchell, and Moggi.

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

Keywords: Polymorphic lambda calculus, type inference, higher-order unification.

1 Introduction

The polymorphic λ-calculus, discovered independently by Girard [6] and Reynolds [21], has served as the basis for many investigations into the nature of polymorphism in programming languages. While it was known that the simply-typed λ-calculus admits principal type schemes and its type inference problem is decidable [10, 16], an analysis of type inference for the polymorphic λ-calculus has proved more difficult.

There appear to be at least two different notions of type inference, both of which are decidable over the simply-typed fragment. One naturally arises from an explicitly typed formulation of the polymorphic λ-calculus in the style of Church, in which terms contain enough types to determine unique types for valid terms. This problem has been called partial type inference by H.-J. Boehm, who showed that, with certain minor additional assumptions, it is undecidable [1]. This result has been sharpened by the author in [19], where it is also argued that the problem can be solved effectively using higher-order unification. The other notion of type inference arises more naturally from a formulation in the style of Curry, in which terms carry no type information at all, and a types may be considered properties of untyped terms. This problem has been called type inference, full type inference, and type reconstruction, and has resisted complete analysis, despite intensive efforts and some partial answers (see, for example, [14, 12, 4]).

We refer to (a variation of) Boehm's problem as *partial type reconstruction* and the other problem as *full type reconstruction*. We believe that partial type reconstruction is the practically more useful problem, and a number of implementations have been based on decidable subcases (see, for example, [2, 20, 11]). Further discussion can be found in [19].

In this paper we prove that partial type reconstruction for the pure polymorphic λ-calculus is undecidable. This proof is a slightly modified version of the one sketched in [19]. Analysis of this proof reveals that the result can be sharpened further in two directions: (1) the problem remains undecidable even if we allow only type variables to occur in a partially typed term, and (2) the problem remains undecidable even in a very simple predicative fragment.

The remainder of this paper is organized as follows. In Section 2 we present an explicitly typed formulation of the pure polymorphic λ-calculus and state some elementary properties. In Section 3 we define the partial and full type reconstruction problems for this calculus. In Section 4 we give a formulation of the second-order unification problem which has been shown to be undecidable by Goldfarb [8], and which we reduce to partial type reconstruction. In Section 5 we develop this reduction and undecidability proof for partial type reconstruction. In Section 6 we show how this result extends to a predicative fragment which is contained in λ^{ML} [9].

2 The Polymorphic λ-Calculus

Variations of second-order polymorphic λ-calculus go back to Girard's system F [5, 6, 7] and Reynolds [21]. Here we treat the pure, type-theoretic core of the language, without recursion or existential types, for example. The undecidability result for this fragment also applies to conservative extensions of this language, that is, extensions which do not affect typability of the pure fragment presented here. Extensions by predefined constants, recursion, exceptions, references, dependent types, functions between types, existential and inductive types would typically be conservative in this sense. On the other hand, the addition of recursive types or conjunctive types would typically not be conservative, since more terms in the core language terms become typable.

The starting point for the partial type reconstruction problem, defined in Section 3, is an explicitly typed calculus, sometimes referred to as a formulation in the style of Church. In Section 3 we will also say more about the relationship to an implicitly typed formulation in the style of Curry in which the terms contain no type information at all.

Our formulation has two language levels: *terms* (denoted by M and N) and *types* (denoted by τ).

$$\begin{array}{llcl} \text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \Delta\alpha.\ \tau \\ \text{Terms} & M & ::= & x \mid \lambda x{:}\tau.\ M \mid M_1\, M_2 \mid \Lambda\alpha.\ M \mid M\,[\tau] \end{array}$$

We let α range over type variables and x, y, z, and sometimes f and g stand for term variables. The typing judgment also requires a notion of *context* which assigns types to free term variables. For technical reasons, we also include declarations for type variables in the context.

$$\text{Context} \quad \Gamma \quad ::= \quad \cdot \mid \Gamma, x{:}\tau \mid \Gamma, \alpha{:}\text{Type}$$

The empty context is denoted by $\cdot$, which we omit on the left-hand side of the typability judgment and at the beginning of context sequences. That is, the context $\cdot, \alpha{:}\text{Type}, x{:}\alpha$ is abbreviated by $\alpha{:}\text{Type}, x{:}\alpha$. To simplify the technical development we assume that no type or term variable is declared in a context more than once. We denote the type assigned to a variable x in a context Γ by $\Gamma(x)$. Additionally, we will tacitly apply α-conversion (renaming of bound variables) at the level of terms and types (where λ, Λ, Δ bind variables). $[\tau'/\alpha]\tau$ denotes the result of substituting τ' for free occurrences of α in τ, renaming bound variables in τ as necessary in order to avoid name clashes. Similary, we write $[\tau/\alpha]\Gamma$ for the result of substituting τ for α in Γ if no free variable in τ is declared in Γ.

The judgments defining typability and validity in this formulation of the polymorphic λ-calculus are

$$\begin{array}{ll} \Gamma \vdash \tau : \text{Type} & \tau \text{ is valid in } \Gamma \\ \Gamma \vdash M : \tau & M \text{ has type } \tau \text{ in } \Gamma \\ \vdash \Gamma \quad \text{Valid} & \Gamma \text{ is valid} \end{array}$$

They are defined by the following sets of inference rules.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\,\text{Var} \qquad \frac{\Gamma \vdash \tau'{:}\text{Type} \qquad \Gamma, x : \tau' \vdash M : \tau}{\Gamma \vdash \lambda x{:}\tau'.\ M : \tau' \to \tau}\,\text{Lam}$$

$$\frac{\Gamma \vdash M : \tau' \to \tau \qquad \Gamma \vdash M' : \tau'}{\Gamma \vdash M\, M' : \tau}\,\text{App} \qquad \frac{\Gamma, \alpha{:}\text{Type} \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.\ M : \Delta\alpha.\ \tau}\,\text{TLam}$$

$$\frac{\Gamma \vdash \tau' : \text{Type} \qquad \Gamma \vdash M : \Delta\alpha.\ \tau}{\Gamma \vdash M\,[\tau'] : [\tau'/\alpha]\tau}\,\text{TApp}$$

The rules Lam and TLam are restricted to the case where x and α, respectively, do not already occur in Γ. In these inference rules we check validity of types in order to ensure that, if Γ is valid (defined below) in the final judgment of a typing derivation, then Γ must be valid throughout the derivation.

$$\frac{\Gamma(\alpha) = \text{Type}}{\Gamma \vdash \alpha : \text{Type}}\,\text{TVar} \qquad \frac{\Gamma \vdash \tau_1 : \text{Type} \qquad \Gamma \vdash \tau_2 : \text{Type}}{\Gamma \vdash \tau_1 \to \tau_2 : \text{Type}}\,\text{Arrow}$$

$$\frac{\Gamma, \alpha{:}\text{Type} \vdash \tau : \text{Type}}{\Gamma \vdash \Delta\alpha.\ \tau : \text{Type}}\,\text{Delta}$$

Validity of contexts reduces to the validity of the types occurring in it.

$$\frac{}{\vdash \cdot \quad \text{Valid}} \qquad \frac{\vdash \Gamma \quad \text{Valid}}{\vdash \Gamma, \alpha{:}\text{Type} \quad \text{Valid}} \qquad \frac{\vdash \Gamma \quad \text{Valid} \qquad \Gamma \vdash \tau : \text{Type}}{\vdash \Gamma, x{:}\tau \quad \text{Valid}}$$

Definition 1 (Validity) *A context* Γ *is* valid *if* $\vdash \Gamma \quad$ Valid *can be derived using the inference rules above. A type* τ *is* valid in context Γ *if* $\Gamma \vdash \tau$: Type *is derivable from the rules above. A term* M *is* valid in context Γ *if* $\Gamma \vdash M : \tau$ *is derivable for some type* τ*, using the rules above.*

In the remainder of the paper we will often abbreviate the phrase of the form "[*a judgment*] is derivable" by "[*a judgment*]." For example, the judgment "$\Gamma \vdash M : \tau$" might stand for the proposition "$\Gamma \vdash M : \tau$ is derivable."

The polymorphic λ-calculus has a number of remarkable properties, such as the Church-Rosser property and strong normalization for valid terms (see, for example, [7]). We will need only a very limited set of properties of the calculus, which means that the main undecidability result also holds in extensions where the stronger properties fail, for example, in an extension by a fixpoint operator.

Proposition 2 (Basic Properties of the Polymorphic λ-Calculus)

1. (Weakening) *Let* M *be a term with no free occurrence of* x*. Then* $\Gamma, x{:}\tau' \vdash M : \tau$ *iff* $\Gamma \vdash M : \tau$.
2. (Uniqueness of Typing Derivations) *Let* Γ *be a valid context and let* M *be a term valid in* Γ*. Then there exists a unique (up to* α*-conversion)* τ *and a unique derivation of* $\Gamma \vdash M : \tau$.
3. (Decidability) *Given a context* Γ *and a term* M*. Then it is decidable whether* Γ *is valid and whether* M *is valid in* Γ.

The proofs of these basic properties are immediate and require only very simple inductions. It is crucial for these properties that terms are explicitly typed.

3 Partial Type Reconstruction

A number of typing problems associated with the polymorphic λ-calculus have been considered in the literature. These have been referred to as type checking, type reconstruction, type inference, and partial type inference, but no standard terminology appears to exist. Thus we will explicitly define various notions, beginning with the notion of a *partially typed term* or *preterm*, denoted by P.

$$\text{Preterms} \quad P \quad ::= \quad x \mid \lambda x{:}\tau.\, P \mid P_1\, P_2 \mid \Lambda\alpha.\, P \mid P\,[\tau] \mid \lambda x.\, P \mid P\,[\,]$$

The partial type reconstruction problem for preterms arises from *partial erasure*, in which types can be omitted, but a "marker" must be left wherever a type has been omitted. Another notion of partial inference has been considered by McCracken [14].

Definition 3 (Partial Erasure) *Let the judgment $P \leq M$ (read: P is a partial erasure of M) be defined by the following inference rules.*

$$\frac{}{x \leq x} \qquad \frac{P \leq M}{\lambda x{:}\tau.\ P \leq \lambda x{:}\tau.\ M} \qquad \frac{P \leq M}{\lambda x.\ P \leq \lambda x{:}\tau.\ M}$$

$$\frac{P_1 \leq M_1 \qquad P_2 \leq M_2}{P_1\, P_2 \leq M_1\, M_2} \qquad \frac{P \leq M}{\Lambda\alpha.\ P \leq \Lambda\alpha.\ M}$$

$$\frac{P \leq M}{P\,[\tau] \leq M\,[\tau]} \qquad \frac{P \leq M}{P\,[\,] \leq M\,[\tau]}$$

Definition 4 (Partial Type Reconstruction) *Given a valid context Γ and a preterm P, determine if there exists a term M valid in Γ such that $P \leq M$. If such an M exists, we call P valid in Γ and write $\Gamma \rhd P$.*

We show in Theorem 23 that partial type reconstruction is undecidable. A similar, but technically weaker result was first reported by H.-J. Boehm [1] and anticipated by Mitchell [17]. Boehm's proof requires a fixpoint operator and an uninterpreted type constant in the language. In view of the undecidability result, restrictions on partially typed terms have been proposed which lead to a decidable type reconstruction problem (see [2, 11]). Our own view is to allow the full range of partially typed terms and use a variant of second-order unification to perform type reconstruction as suggested in [19].

More difficult to analyze than partial type reconstruction has been the problem of *full type reconstruction.* In our framework, this problem can be characterized if we introduce *untyped terms.*

$$\text{Untyped Terms} \quad U \quad ::= \quad x \mid \lambda x.\, U \mid U_1\, U_2$$

The erasure relation now becomes simpler.

Definition 5 (Full Erasure) *Let the judgment $U \prec M$ (read: U is the full erasure of M) be defined by the following inference rules.*

$$\frac{}{x \prec x} \qquad \frac{U \prec M}{\lambda x.\, U \prec \lambda x{:}\tau.\ M} \qquad \frac{U \prec M \qquad V \prec N}{U\,V \prec M\,N}$$

$$\frac{U \prec M}{U \prec \Lambda\alpha.\ M} \qquad \frac{U \prec M}{U \prec M\,[\tau]}$$

Definition 6 (Full Type Reconstruction) *Given a valid Γ and an untyped term U, determine if there exists a term M valid in Γ such that $U \prec M$.*

The decidability of full type reconstruction is still open, despite intensive efforts and a number of partial results (see, for example, [12, 4]). Unfortunately, our undecidability results seems to bear no direct relationship to the full type reconstruction problem, nor do we see how our techniques could be applied.

While one might feel that full type reconstruction is a more fundamental, mathematical problem, it seems to us that partial type reconstruction is a more useful problem in the context of realistic programming languages, when augmented with *type argument synthesis*, which is an orthogonal issue and beyond the scope of this paper. Further discussion on this issue can be found in [19]. In particular, we indicate how it could be considered a natural generalization of type inference in the Damas-Milner calculus [3] which is the basis for type inference in the programming language ML.

To illustrate the difference between partial and full type reconstruction, consider the preterm $\lambda x.\, x\,[\,]\,x$ in the empty context. It can easily be checked that, for example,

$$\lambda x.\, x\,[\,]\,x \leq \lambda x{:}\Delta\alpha.\, \alpha \to \alpha.\, x\,[\Delta\alpha.\, \alpha \to \alpha]\,x \qquad (= M_1)$$

and

$$\lambda x.\, x\,[\,]\,x \leq \lambda x{:}\Delta\alpha.\, \alpha.\, x\,[(\Delta\alpha.\, \alpha) \to \tau]\,x \qquad (= M_2)$$

for any valid type τ. Note that M_1 and M_2 are both valid in the empty context. On the other hand, there does not exist a valid M such that

$$\lambda x.\, x\,x \leq M$$

while both

$$\lambda x.\, x\,x \prec M_1 \qquad \text{and} \qquad \lambda x.\, x\,x \prec M_2$$

hold. Note that in the simply-typed λ-calculus the problems of partial and full type reconstruction are both decidable and can be solved with essentially the same algorithm based on (first-order) unification.

Independently of the question of decidability, this example also shows partial type reconstruction does not have the principal type property. That is, for a preterm P there may be many different valid terms M_i and types τ_i such that $P \leq M_i$ and $\vdash M_i : \tau_i$, but the τ_i may *not* be instances of a common type schema τ (all of whose instances are types of P).

4 A Unification Logic

We would like to show the undecidability of the partial type reconstruction problem by a reduction from the unifiability problem of the second-order[1] fragment of the simply-typed λ-calculus. This problem has been shown to be undecidable in the presence of at least one binary function constant by Goldfarb [8].

In order to simplify the reduction we define a variant of second-order unification which can easily be seen as a generalization of the standard formulation. A related formulation in terms of *mixed prefixes* is given by Miller [15].

The basic notion of the unification logic is that of a *formula*, and unifiability is replaced by provability of a formula, as defined below. The basic formulas are equations between *types*, including variables ranging over functions between types. This requires types and functions between them to be classified by *kinds* and they thus form a "simply-kinded" λ-calculus. We use F to stand for formulas in the unification logic. Moreover, we use α for type variables, β for type variables which may range over type functions, and σ for types and functions between types (which we call *extended types*).

$$\begin{array}{rrcl}
\text{Kinds} & K & ::= & \text{Type} \mid \text{Type} \to K \\
\text{Extended Types} & \sigma & ::= & \alpha \mid \beta \mid \lambda\alpha.\, \sigma \mid \sigma_1\,\sigma_2 \mid \sigma_1 \to \sigma_2 \mid \Delta\alpha.\, \sigma \\
\text{Extended Contexts} & \Psi & ::= & \cdot \mid \Psi, \alpha{:}\text{Type} \mid \Psi, \beta{:}K \mid \Psi, x{:}\sigma \\
\text{Formulas} & F & ::= & \sigma_1 \doteq \sigma_2 \mid F_1 \wedge F_2 \mid \top \mid \exists\beta{:}K.\, F \mid \forall\alpha.\, F
\end{array}$$

[1]This notion of "second-order" is not to be confused with the "second-order" as it appears in the phrase "second-order polymorphic λ-calculus."

The restriction to second order is incorporated directly into this formulation by restricting function kinds to have domain Type, rather than allowing the more general form $K_1 \to K_2$. We will drop the by-word "extended" if it is clear that we are referring to an extended type or context. Validity of extended types is defined as in a simply-typed λ-calculus, except that kinds K play the role ordinarily played by types, and extended types σ play the role of terms. The rules for valid types in Section 2 carry over and the following new rules are added.

$$\frac{\Psi(\beta) = K}{\Psi \vdash \beta : K} \qquad \frac{\Psi, \alpha{:}\text{Type} \vdash \sigma : K}{\Psi \vdash \lambda\alpha.\, \sigma : \text{Type} \to K} \qquad \frac{\Psi \vdash \sigma_1 : \text{Type} \to K \qquad \Psi \vdash \sigma_2 : \text{Type}}{\Psi \vdash \sigma_1\, \sigma_2 : K}$$

We write $\sigma_1 =_{\beta\eta} \sigma_2$ if σ_1 and σ_2 are $\beta\eta$-convertible in the usual sense, and $[\sigma/\alpha]F$ stands for the result of substituting σ for α in F, renaming bound variables (including those bound by $\exists$ and $\forall$) to avoid name clashes. Provability in the unification logic is defined through the following inference rules. Note that we restrict ourselves here to ordinary contexts Γ, containing no declarations of type functions. This is possible because such type functions may occur only existentially quantified and thus never enter the context in a derivation establishing provability of a closed formula.

$$\frac{\Gamma \vdash \sigma_1 : \text{Type} \qquad \sigma_1 =_{\beta\eta} \sigma_2 \qquad \Gamma \vdash \sigma_2 : \text{Type}}{\Gamma \Vdash \sigma_1 \doteq \sigma_2} \qquad \frac{\Gamma \Vdash F_1 \qquad \Gamma \Vdash F_2}{\Gamma \Vdash F_1 \wedge F_2} \qquad \frac{}{\Gamma \Vdash \top}$$

$$\frac{\Gamma \vdash \sigma : K \qquad \Gamma \Vdash [\sigma/\beta]F}{\Gamma \Vdash \exists\beta{:}K.\, F} \qquad \frac{\Gamma, \alpha{:}\text{Type} \Vdash F}{\Gamma \Vdash \forall\alpha.\, F}$$

The following proposition is obvious from the set of inference rules.

Proposition 7 (Inversion) *Let Γ be a valid context and F be formula. If F is provable in Γ, then the last inference rule in the derivation of $\Gamma \Vdash F$ is uniquely determined.*

A well-formed second-order unification problem can be reduced a theorem proving problem in the unification logic as follows. Let

$$\{\sigma_1 \doteq \sigma_1', \ldots, \sigma_n \doteq \sigma_n'\}$$

be a set of second-order equations (considering "$\to$" as a single binary function constant) whose free variables are $\beta_1, \ldots, \beta_m$ of kinds $K_1, \ldots, K_m$, respectively. The interpretation of this set as a formula is defined as

$$\exists\beta_1{:}K_1 \ldots \exists\beta_m{:}K_m.\, |\sigma_1 \doteq \sigma_1'| \wedge \ldots \wedge |\sigma_n \doteq \sigma_n'|$$

where

$$|\sigma \doteq \sigma'| = \forall\alpha_1 \ldots \forall\alpha_k.\, \sigma\, \alpha_1 \ldots \alpha_k \doteq \sigma'\, \alpha_1 \ldots \alpha_k$$

if σ and σ' have kind

$$\underbrace{\text{Type} \to \cdots \to}_{k} \text{Type}$$

For example, $|(\lambda\alpha.\, \alpha) \doteq (\lambda\alpha.\, \beta\, \alpha)| = \exists\beta{:}\text{Type} \to \text{Type}.\, \forall\alpha.\, \alpha \to \alpha \doteq \beta\, \alpha$.

Theorem 8 *Given a formula F. Then provability of F is undecidable even if F contains no occurrence of Δ.*

Proof: Goldfarb [8] showed that the second-order unification problem is undecidable in the presence of at least one binary function constant.

The reduction of this problem to the provability problem in the unification logic above is straightforward following the notes above and Miller [15]. "$\rightarrow$" plays the role of the required binary function constant; Δ is not required in order to attain undecidability. $\square$

We also need a notion of type substitution in order to carry out the proofs in Section 3. In this definition we need to traverse the context from left to right in order to properly account for the scope of type variables in a context.

Definition 9 (Type Substitution) *Let Ψ be an valid extended context. A Ψ-*substitution *S has the form $[\beta_1 \mapsto \sigma_1, \ldots, \beta_n \mapsto \sigma_n]$ such that $\beta_1, \ldots, \beta_n$ (the* domain *of S) are variables declared in Ψ. S is called* valid *if $S\,\Psi$ is a valid context, defined by*

$$\begin{array}{rcll} S\,\cdot & = & \cdot & \\ S(\alpha{:}\mathrm{Type}, \Psi) & = & \alpha{:}\mathrm{Type}, S\Psi & \textit{if } \alpha \textit{ not in the domain of } S \\ S(\beta{:}K, \Psi) & = & \beta{:}K, S\Psi & \textit{if } \beta \textit{ not in the domain of } S \\ S(\beta{:}K, \Psi) & = & S([\sigma/\beta]\Psi) & \textit{if } [\beta \mapsto \sigma] \textit{ in } S \\ S(x{:}\sigma, \Psi) & = & x{:}S\sigma, S\Psi & \end{array}$$

Here $S\sigma$ stands for the usual application of a substitution S to a type σ, renaming bound type variables in order to avoid name clashes. Similarly, SF stands for the result of applying the substitution S to the formula F. The extension of a substitution S is written as $S \oplus [\beta \mapsto \sigma]$, where β may not already appear in the domain of S.

Thus, for example, $[\beta \mapsto \alpha \rightarrow \alpha](\alpha{:}\mathrm{Type}, \beta{:}\mathrm{Type}, x{:}\beta) = \alpha{:}\mathrm{Type}, x{:}\alpha \rightarrow \alpha$. Although it is by no means necessary in general, for the purposes of this paper it is convenient to restrict attention to Ψ-substutitions S such that ΨS is a valid context without type functions.

Proposition 10 (Elementary Properties of Substitution) *Let S be a valid Ψ-substitution and let σ be an extended type such that $S\Psi \vdash \sigma : K$. Then $S \oplus [\beta \mapsto \sigma]$ is a valid $(\Psi, \beta{:}K)$-substitution and $(S \oplus [\beta \mapsto \sigma])F = [\sigma/\beta](SF)$.*

5 Undecidability of Partial Type Reconstruction

In this section we prove the undecidability of partial type reconstruction from Definition 4. This is achieved via a translation of formulas in the unification logic to preterms, such that the formula is provable iff the resulting preterm is valid (typable).

The first lemma is a central but straightforward observation. In the full type reconstruction problem, there appears to be no way to formulate a corresponding lemma—thus the technique shown here does not seem to help in dealing with full type reconstruction.

Lemma 11 (Forcing Type Equality) *Let Γ be a valid context, and let P_1, P_2, and P be preterms with no free occurrences of the variable f. Then $\Gamma \triangleright \lambda f.\ f\,P_1\,(f\,P_2\,P)$ iff $\Gamma \triangleright P$ and there exist terms M_1 and M_2 and a type τ such that $P_1 \leq M_1$, $P_2 \leq M_2$, $\Gamma \vdash M_1 : \tau$, and $\Gamma \vdash M_2 : \tau$.*

Proof: First assume that $\Gamma \triangleright \lambda f.\ f\,P_1\,(f\,P_2\,P)$. Then there exists a term N valid in Γ such that $\lambda f.\ f\,P_1\,(f\,P_2\,P) \leq N$. From the inference rules for $\leq$ we know that N must have the form $\lambda f{:}\tau'.\ f\,M_1\,(f\,M_2\,M)$. Since N was assumed to be valid, we can construct a unique typing derivation for N (see Proposition 2), which is determined by the structure of N. By inspecting this derivation we can see that it must contain a subderivation of $\Gamma, f{:}\tau' \vdash M : \tau''$ for some τ''. Furthermore, it must contain subderivations of $\Gamma, f{:}\tau' \vdash M_1 : \tau$ and $\Gamma, f{:}\tau' \vdash M_2 : \tau$ for some τ and $\tau' = \tau \to \tau'' \to \tau''$. Since P_1, P_2, and P and therefore M_1, M_2, and M do not contain free occurrences of f, we conclude that $\Gamma \vdash M_1 : \tau$ and $\Gamma \vdash M_2 : \tau$.

For the other direction we simply have to construct a small typing derivation of $\Gamma \vdash \lambda f{:}\tau \to \tau'' \to \tau''.\ f\, M_1\, (f\, M_2\, M)$, using the derivations of $\Gamma \vdash M_1 : \tau$, $\Gamma \vdash M_2 : \tau$, and $\Gamma \vdash M : \tau''$. $\square$

The pairing lemma allows the pairing of subproblems which might arise in the course of the reduction, where their interaction is limited to common variables.

Lemma 12 (Pairing) *Let Γ be a valid context and $P_1, \ldots, P_n$ be preterms with no free occurrences of the variable g. Then $\Gamma \rhd \lambda g.\ g\, P_1 \ldots P_n$ iff $\Gamma \rhd P_i$ for all $1 \le i \le n$.*

Proof: Immediate, following simple reasoning as in the proof of Lemma 11. $\square$

Lemma 14 establishes that, given an arbitrary type τ, we can create a preterm P with one free variable x such that P is valid iff x is assigned the type τ (up to α-conversion between types, of course).

Definition 13 (Mapping $\lfloor x \rfloor_\Gamma^\tau$) *Let Γ be a valid context and τ a valid type in Γ. We define the preterm $\lfloor x \rfloor_\Gamma^\tau$ by induction on the structure of τ.*

Case: $\tau = \alpha$. *Then*

$$\lfloor x \rfloor_\Gamma^\alpha = \lambda z{:}\alpha.\ \lambda f.\ f\, x\, (f\, z\, (\lambda g.\ g))$$

Case: $\tau = \tau_1 \to \tau_2$. *Then*

$$\lfloor x \rfloor_\Gamma^{\tau_1 \to \tau_2} = \lambda z_1.\ \lambda z_2.\ \lambda f.\ f\, (x\, z_1)\, (f\, z_2\, (\lambda g.\ g\, (\lfloor z_1 \rfloor_\Gamma^{\tau_1})\, (\lfloor z_2 \rfloor_\Gamma^{\tau_2})))$$

Case: $\tau = \Delta\alpha.\ \tau_1$. *Then*[2]

$$\lfloor x \rfloor_\Gamma^{\Delta\alpha.\ \tau_1} = \Lambda\alpha.\ \lambda z_1.\ \lambda f.\ f\, (x\, [\alpha])\, (f\, z_1\, (\lambda g.\ g\, (\lfloor z_1 \rfloor_{\Gamma,\alpha:\text{Type}}^{\tau_1})))$$

We will not need the following lemma directly, but its proof is instructive, as the proof of the crucial Lemma 19 proceeds by a similar argument.

Lemma 14 (Forcing Types) *Given a valid context Γ and a type τ valid in Γ. Then $\Gamma, x{:}\tau' \rhd \lfloor x \rfloor_\Gamma^\tau$ iff $\tau' = \tau$ (up to α-conversion).*

Proof: The proof is a straightforward induction over the structure of τ, using Lemmas 11 and 12.

The case of $\tau = \alpha$ follows immediately from Lemma 11.

In the case for $\tau_1 \to \tau_2$ we know by induction hypothesis and Lemma 12 that z_1 must be assigned type τ_1 and z_2 must be assigned type τ_2. The sub-preterm $(x\, z_1)$ forces x to be of function type with domain τ_1, the type of z_1. The range type of x must be equal to the type of z_2 (by Lemma 11) and thus τ_2.

Similarly, in the case of $\Delta\alpha.\ \tau_1$, we know by induction hypothesis that z_1 must have type τ_1 in context $\Gamma, \alpha{:}$Type. The sub-preterm $(x\, [\alpha])$ forces x to be of type $\Delta\alpha.\ \tau_1'$. The type of this type application, $[\alpha/\alpha]\tau_1' = \tau_1'$, must be equal to the type of z_1 by Lemma 11, and thus $\tau_1' = \tau_1$. $\square$

[2]The abstraction over g is redundant here, and inserted only for symmetry with the other cases.

In the formula translation in Definition 20, we have to consider variables which are "existential" and can not be mentioned in the preterm we are constructing. Moreover, some of these variables might be of second order, that is, type functions. We thus extend the previous translation and lemma to allow for these.

Definition 15 (Type Closure) *Let* β:Type $\to \cdots \to$ Type *be a type variable. Then* $\overline{\beta}$*, the* closure *of* β*, is defined by* $\overline{\beta} = \Delta\alpha_1 \ldots \Delta\alpha_n.\ \alpha_1 \to \cdots \to \alpha_n \to \beta\,\alpha_1 \ldots \alpha_n$.

One of the basic ideas in the translation from formulas to term is that an existentially quantified variable β in a formula F corresponds to an omitted type in a preterm. That is, $\exists\beta$:Type. F is translated to a preterm $\lambda x.\ P$, where P is the result of translating F. Where the type variable β occurs in F, we use the variable x in P in such a way that the constraints imposed by the equations in F are equivalent to the constraints on the type of x in P. Thus we need to maintain a mapping from type variables in F to term variables in P. It is convenient to maintain this mapping in a context of a special form, an *invertible context*. The definition is complicated slightly by type functions. If an existentially quantified type variable β is a type function, we arrange that the corresponding term variable has the type of the closure of β.

Definition 16 (Invertible Contexts) *An extended context* Ψ *is called* invertible *if for each type variable* β *declared in* Ψ *there exists a unique term variable* x *such that* $\Psi(x) = \overline{\beta}$*. If* Ψ *is invertible, we denote the unique variable* x *such that* $\Psi(x) = \overline{\beta}$ *by* $\Psi^{-1}(\beta)$.

Lemma 17 (Basic Property of Invertible Contexts) *Given a valid invertible context* Ψ *and a valid* Ψ*-substitution* S*. Then* $S\Psi \vdash \Psi^{-1}(\beta) : S\overline{\beta}$.

Definition 18 (Mapping $\|x\|^{\sigma}_{\Psi}$) *Let* Ψ *be a valid invertible context and* σ *an extended type (of kind* Type*) valid in* Ψ*. We define a preterm* $\|x\|^{\sigma}_{\Psi}$ *by induction on the structure of* σ*. The first case is a degenerate subcase of the second, exhibiting the basis for this inductive definition.*

Case: $\sigma = \alpha$*. Then*

$$\|x\|^{\alpha}_{\Psi} = \lambda f.\ f\,x\,(f\,(\Psi^{-1}(\alpha))\,(\lambda g.\ g))$$

Case: $\sigma = \beta\,\sigma_1 \ldots \sigma_n$*. Then*

$$\|x\|^{\beta\sigma_1\ldots\sigma_n}_{\Psi} = \lambda z_1 \ldots \lambda z_n.\ \lambda f.\ f\,x\,(f\,((\Psi^{-1}(\beta))\,[\,]\ldots[\,]\,z_1 \ldots z_n)\,(\lambda g.\ g\,(\|z_1\|^{\sigma_1}_{\Psi})\ldots(\|z_n\|^{\sigma_n}_{\Psi})))$$

Case: $\sigma = \sigma_1 \to \sigma_2$*. Then*

$$\|x\|^{\sigma_1\to\sigma_2}_{\Psi} = \lambda z_1.\ \lambda z_2.\ \lambda f.\ f\,(x\,z_1)\,(f\,z_2\,(\lambda g.\ g\,(\|z_1\|^{\sigma_1}_{\Psi})\,(\|z_2\|^{\sigma_2}_{\Psi})))$$

Case: $\sigma = \Delta\alpha.\ \sigma_1$*. Then*

$$\|x\|^{\Delta\alpha.\,\sigma_1}_{\Psi} = \Lambda\alpha.\ \lambda z_1.\ \lambda f.\ f\,(x\,[\alpha])\,(f\,z_1\,(\lambda g.\ g\,(\|z_1\|^{\sigma_1}_{\Psi,\alpha:\mathrm{Type},z_1:\alpha})))$$

For this mapping we need a stronger property than Lemma 14. We need to guarantee that the type variables declared in Ψ do not occur in $\|x\|^{\sigma}_{\Psi}$. This is necessary, since $\exists\beta$:Type. F will be translated to $\lambda x.\ P$, where the type of x and the instantation for β will be forced to correspond. But the type variable β itself can not be mentioned in M, because the type assigned to x must remain unspecified. This property is embodied in next lemma by requiring that $\|x\|^{\sigma}_{\Psi}$ must be valid under any valid Ψ-substitution.

Lemma 19 (Forcing Types) *Given a valid invertible context Ψ, a valid Ψ-substitution S, and an extended type σ valid in Ψ. Then, for any valid type τ,*

$$S\Psi, x{:}\tau \rhd \|x\|^{\sigma}_{\Psi} \quad \textit{iff} \quad S\sigma =_{\beta\eta} \tau.$$

Moreover, $S\Psi, x{:}S\sigma \rhd \|x\|^{\sigma}_{\Psi}$ *iff* $S\Psi \rhd \lambda x.\ \|x\|^{\sigma}_{\Psi}$.

Proof: The second part of the Lemma is an easy consequence of the first part. The proof of the first part proceeds by induction on the structure of σ, where we take advantage of the second part for the induction hypothesis. We implicitly rely on some elementary reasoning about typing derivations as in the proof of Lemma 14.

Case: $\sigma = \alpha$. Then

$$\begin{array}{lll} & S\Psi, x{:}\tau \rhd \|x\|^{\sigma}_{\Psi} & \\ \text{iff} & S\Psi, x{:}\tau \rhd \lambda f.\ f\,x\,(f\,(\Psi^{-1}(\alpha))\,(\lambda g.\ g)) & \text{by definition} \\ \text{iff} & S\Psi, x{:}\tau \vdash \Psi^{-1}(\alpha) : \tau' \quad \text{and} \quad \tau' =_{\beta\eta} \tau & \text{by Lemma 11} \\ \text{iff} & \tau' = S\alpha =_{\beta\eta} \tau & \text{by Lemma 17} \end{array}$$

Case: $\sigma = \beta\,\sigma_1 \ldots \sigma_n$. Then

$$\begin{array}{lll} & S\Psi, x{:}\tau \rhd \|x\|^{\sigma}_{\Psi} & \\ \text{iff} & S\Psi, x{:}\tau \rhd \lambda z_1 \ldots \lambda z_n.\ \lambda f.\ f\,x\,(f\,((\Psi^{-1}(\beta))\,[\,]\ldots[\,]\,z_1\ldots z_n)(\lambda g.\ g\,(\|z_1\|^{\sigma_1}_{\Psi})\ldots(\|z_n\|^{\sigma_n}_{\Psi}))) & \\ \text{iff} & S\Psi, x{:}\tau, z_1{:}S\sigma_1, \ldots, z_n{:}S\sigma_n & \\ & \quad \rhd \lambda f.\ f\,((\Psi^{-1}(\beta))\,[\,]\ldots[\,]\,z_1\ldots z_n)\,(f\,x\,(\lambda g.\ g\,(\|z_1\|^{\sigma_1}_{\Psi})\ldots(\|z_n\|^{\sigma_n}_{\Psi}))) & \text{by ind. hyp.} \\ \text{iff} & (S\beta)\,(S\sigma_1)\ldots(S\sigma_n) = S(\beta\,\sigma_1\ldots\sigma_n) =_{\beta\eta} \tau & \text{by Lemma 11} \end{array}$$

Case: $\sigma = \sigma_1 \to \sigma_2$. Then

$$\begin{array}{lll} & S\Psi, x{:}\sigma' \rhd \|x\|^{\sigma_1\to\sigma_2}_{\Psi} & \\ \text{iff} & S\Psi, x{:}\tau \rhd \lambda z_1.\ \lambda z_2.\ \lambda f.\ f\,(x\,z_1)\,(f\,z_2\,(\lambda g.\ g\,(\|z_1\|^{\sigma_1}_{\Psi})\,(\|z_2\|^{\sigma_2}_{\Psi}))) & \text{by definition} \\ \text{iff} & S\Psi, x{:}\tau, z_1{:}S\sigma_1, z_2{:}S\sigma_2 \rhd \lambda f.\ f\,(x\,z_1)\,(f\,z_2\,(\lambda g.\ g\,(\|z_1\|^{\sigma_1}_{\Psi})\,(\|z_2\|^{\sigma_2}_{\Psi}))) & \text{by ind. hyp.} \\ \text{iff} & S\sigma_1 \to S\sigma_2 = S(\sigma_1 \to \sigma_2) =_{\beta\eta} \tau & \text{by Lemma 11} \end{array}$$

Case: $\sigma = \Delta\alpha.\ \sigma_1$. Then

$$\begin{array}{lll} & S\Psi, x{:}\tau \rhd \|x\|^{\Delta\alpha.\ \sigma_1}_{\Psi} & \\ \text{iff} & S\Psi, x{:}\tau \rhd \Lambda\alpha.\ \lambda z_1.\ \lambda f.\ f\,(x\,[\alpha])\,(f\,z_1\,(\lambda g.\ g\,(\|z_1\|^{\sigma_1}_{\Psi,\alpha:\text{Type},z_1:\alpha}))) & \text{by definition} \\ \text{iff} & S\Psi, x{:}\tau, \alpha{:}\text{Type}, z_1{:}S\sigma_1 \rhd \lambda f.\ f\,(x\,[\alpha])\,(f\,z_1\,(\lambda g.\ g\,(\|z_1\|^{\sigma_1}_{\Psi,\alpha:\text{Type},z_1:\alpha}))) & \\ & & \text{by ind. hyp.} \\ \text{iff} & \Delta\alpha.\ S\sigma_1 = S(\Delta\alpha.\ \sigma_1) =_{\beta\eta} \tau & \text{by Lemma 11} \end{array}$$

$\square$

Now we come to the main part of the undecidability proof: a translation from formulas to preterms, mapping provability to validity. It follows the ideas discussed informally above.

Definition 20 (Formula Translation) *Let Ψ be a valid invertible context. Then we define the preterm $\lceil F \rceil^{\Psi}$ by induction on the structure of F.*

Case: $F = \sigma_1 \doteq \sigma_2$. *Then*

$$\lceil \sigma_1 \doteq \sigma_2 \rceil^\Psi = \lambda z_1.\ \lambda z_2.\ \lambda f.\ f\, z_1\, (f\, z_2\, (\lambda g.\ g\, (\|z_1\|^{\sigma_1}_{\Psi})\, (\|z_2\|^{\sigma_2}_{\Psi})))$$

Case: $F = F_1 \wedge F_2$. *Then*

$$\lceil F_1 \wedge F_2 \rceil^\Psi = \lambda g.\ g\, (\lceil F_1 \rceil^\Psi)\, (\lceil F_2 \rceil^\Psi)$$

Case: $F = \top$. *Then*

$$\lceil \top \rceil^\Psi = \lambda g.\ g$$

Case: $F = \forall \alpha{:}\text{Type}.\ F_1$. *Then*

$$\lceil \forall \alpha{:}\text{Type}.\ F_1 \rceil^\Psi = \Lambda \alpha.\ \lambda x{:}\alpha.\ \lceil F_1 \rceil^{\Psi, \alpha:\text{Type}, x:\alpha}$$

Case: $F = \exists \beta{:}K.\ F_1$. *Then*

$$\lceil \exists \beta{:}\text{Type}.\ F_1 \rceil^\Psi = \lambda x.\ \lambda g.\ g\, (\|x\|^{\overline{\beta}}_{\Psi, \beta:\text{Type}, x:\overline{\beta}})\, (\lceil F_1 \rceil^{\Psi, \beta:\text{Type}, x:\overline{\beta}})$$

Theorem 21 (Reduction of Provability to Partial Type Reconstruction) *Given a valid, invertible context Ψ, a formula F with free variables declared in Ψ, and a valid Ψ-substitution S. Then $S\Psi \Vdash SF$ iff $S\Psi \triangleright \lceil F \rceil^\Psi$.*

Proof: The proof proceeds by induction on the structure of F. The lines not directly justified follow directly from elementary properties of provability, substitution, and validity.

Case: $F = \sigma_1 \doteq \sigma_2$.

Examining the left-hand side of biconditional in the claim yields

$$\begin{array}{ll} & S\Psi \Vdash S(\sigma_1 \doteq \sigma_2) \\ \text{iff} & S\Psi \vdash S\sigma_1 : \text{Type} \quad \text{and} \quad S\Psi \vdash S\sigma_2 : \text{Type} \quad \text{and} \quad S\sigma_1 =_{\beta\eta} S\sigma_2 \end{array}$$

Examining the right-hand side yields

$$\begin{array}{lll} & S\Psi \triangleright \lceil \sigma_1 \doteq \sigma_2 \rceil^\Psi & \\ \text{iff} & S\Psi \triangleright \lambda z_1.\ \lambda z_2.\ \lambda f.\ f\, z_1\, (f\, z_2\, (\lambda g.\ g\, (\|z_1\|^{\sigma_1}_{\Psi})\, (\|z_2\|^{\sigma_2}_{\Psi}))) & \text{by definition} \\ \text{iff} & S\Psi, z_1{:}S\sigma_1, z_2{:}S\sigma_2 \triangleright \lambda f.\ f\, z_1\, (f\, z_2\, (\lambda g.\ g\, (\|z_1\|^{\sigma_1}_{\Psi})\, (\|z_2\|^{\sigma_2}_{\Psi}))) & \text{by Lemmas 12 and 19} \\ \text{iff} & S\sigma_1 =_{\beta\eta} S\sigma_2 \quad \text{and} \quad S\sigma_1 \text{ and } S\sigma_2 \text{ are valid in } S\Psi & \text{by Lemma 11} \end{array}$$

Hence the left-hand and right-hand sides of the theorem are equivalent in this case.

Case: $F = F_1 \wedge F_2$. Then

$$\begin{array}{lll} & S\Psi \Vdash S(F_1 \wedge F_2) & \\ \text{iff} & S\Psi \Vdash SF_1 \wedge SF_2 & \\ \text{iff} & S\Psi \Vdash SF_1 \quad \text{and} \quad S\Psi \Vdash SF_2 & \text{by Proposition 7} \\ \text{iff} & S\Psi \triangleright \lceil F_1 \rceil^\Psi \quad \text{and} \quad S\Psi \triangleright \lceil F_2 \rceil^\Psi & \text{by induction hypothesis} \\ \text{iff} & S\Psi \triangleright \lambda g.\ g\, (\lceil F_1 \rceil^\Psi)\, (\lceil F_2 \rceil^\Psi) & \text{by Lemma 12} \\ \text{iff} & S\Psi \triangleright \lceil F_1 \wedge F_2 \rceil^\Psi & \text{by definition} \end{array}$$

Case: $F = \top$. Then $S\Psi \Vdash \top$ and also $S\Psi \triangleright \lambda g.\ g$.

Case: $F = \exists\beta{:}K.\ F_1$. Then

$$\begin{array}{lll}
& S\Psi \Vdash S(\exists\beta{:}K.\ F_1) & \\
\text{iff} & S\Psi \Vdash \exists\beta{:}K.\ SF_1 & \text{(possibly after renaming)} \\
\text{iff} & S\Psi \Vdash [\sigma/\beta]SF_1 \quad \text{for some } \sigma & \text{by Proposition 7} \\
\text{iff} & (S \oplus [\beta \mapsto \sigma])(\Psi, \beta{:}K, x{:}\overline{\beta}) \Vdash (S \oplus [\beta \mapsto \sigma])F_1 & \text{by Propositions 10 and 2} \\
\text{iff} & (S \oplus [\beta \mapsto \sigma])(\Psi, \beta{:}K, x{:}\overline{\beta}) \rhd \lceil F_1 \rceil^{\Psi,\beta:K,x:\overline{\beta}} & \text{by induction hypothesis} \\
\text{iff} & (S \oplus [\beta \mapsto \sigma])\Psi, x{:}\overline{\sigma} \rhd \lceil F_1 \rceil^{\Psi,\beta:K,x:\overline{\beta}} & \text{by Definition 9} \\
\text{iff} & (S \oplus [\beta \mapsto \sigma])\Psi \rhd \lambda x.\ \lambda g.\ g\,(\|x\|^{\overline{\beta}}_{\Psi,\beta:\mathrm{Type},x:\overline{\beta}})\,(\lceil F_1 \rceil^{\Psi,\beta:K,x:\overline{\beta}}) & \text{by Lemmas 12 and 19} \\
\text{iff} & S\Psi \rhd \lambda x.\ \lambda g.\ g\,(\|x\|^{\overline{\beta}}_{\Psi,\beta:\mathrm{Type},x:\overline{\beta}})\,(\lceil F_1 \rceil^{\Psi,\beta:K,x:\overline{\beta}}) & \\
\text{iff} & S\Psi \rhd \lceil \exists\beta{:}K.\ F_1 \rceil^{\Psi} & \text{by definition}
\end{array}$$

Case: $F = \forall\alpha.\ F_1$. Then

$$\begin{array}{lll}
& S\Psi \Vdash S(\forall\alpha.\ F_1) & \\
\text{iff} & S\Psi \Vdash \forall\alpha.\ SF_1 & \text{(possibly after renaming)} \\
\text{iff} & S\Psi, \alpha{:}\mathrm{Type} \Vdash SF_1 & \text{by Proposition 7} \\
\text{iff} & S\Psi, \alpha{:}\mathrm{Type}, x{:}\alpha \Vdash SF_1 & \\
\text{iff} & S(\Psi, \alpha{:}\mathrm{Type}, x{:}\alpha) \Vdash SF_1 & \\
\text{iff} & S(\Psi, \alpha{:}\mathrm{Type}, x{:}\alpha) \rhd \lceil F_1 \rceil^{\Psi,\alpha:\mathrm{Type},x:\alpha} & \text{by induction hypothesis} \\
\text{iff} & S\Psi, \alpha{:}\mathrm{Type}, x{:}\alpha \rhd \lceil F_1 \rceil^{\Psi,\alpha:\mathrm{Type},x:\alpha} & \\
\text{iff} & S\Psi \rhd \Lambda\alpha.\ \lambda x{:}\alpha.\ \lceil F_1 \rceil^{\Psi,\alpha:\mathrm{Type},x:\alpha} & \\
\text{iff} & S\Psi \rhd \lceil \forall\alpha.\ F_1 \rceil^{\Psi} & \text{by definition}
\end{array}$$

□

Corollary 22 *Let F be a closed formula. Then $\Vdash F$ iff $\rhd \lceil F \rceil$.*

Theorem 23 (Undecidability of Partial Type Reconstruction) *In the pure polymorphic λ-calculus, the problem of partial type reconstruction is undecidable.*

Proof: The problem of provability in the unification logic is undecidable (see Theorem 8). Since provability can be reduced to partial type reconstruction by Corollary 22, partial type reconstruction is undecidable. □

The range of the mapping $\lceil F \rceil$ mentions only type variables α. Therefore we can strengthen Theorem 23 further by considering a class of preterms containing only type variables, and those only in abstractions.

$$Q \quad ::= \quad x \mid \lambda x{:}\alpha.\ Q \mid Q_1\,Q_2 \mid \Lambda\alpha.\ Q \mid \lambda x.\ Q \mid Q\,[\,]$$

Corollary 24 *The partial type reconstruction problem for preterms of the form Q is undecidable.*

Proof: Let F be closed formula without occurrences of Δ. Then $\lceil F \rceil$ has the form of Q above. By Corollary 22, F is provable iff $\lceil F \rceil$ is valid. But provability of F is undecidable, and hence the limited form of type reconstruction is also undecidable. □

Since in any practical language one would like to allow user-specified type annotations, we do not consider this corollary to be particularly important. We have not investigated the question if partial type reconstruction would be undecidable even for terms completely devoid of types (except for placeholders $[\,]$ and abstractions $\Lambda\alpha$).

Another straightforward observation is that preterms of the form $\lceil F \rceil$ are in normal form.

Corollary 25 *The partial type reconstruction problem for preterms of the form Q which are also in β-normal form is undecidable.*

6 Partial Type Reconstruction in Predicative Fragments

One might conjecture that the undecidability of type reconstruction is due to the inherent expressive power of the pure polymorphic λ-calculus. However, this is not the case—even a very simple predicative fragment has an undecidable partial type reconstruction problem. This can be seen by carefully examining the proofs showing the undecidability of partial type reconstruction in the polymorphic λ-calculus given in Section 5.[3] This means that partial type reconstruction for λ^{ML} (see [18, 9]) is also undecidable.

The polymorphic λ-calculus is impredicative in that the domain of quantification (by Δ) includes all possible types. The term $M_1 = \lambda x{:}\Delta\alpha.\ \alpha \to \alpha.\ x\,[\Delta\alpha.\ \alpha \to \alpha]\,x$ introduced in the example at the end of Section 3 illustrates this impredicativity: $\Delta\alpha.\ \alpha \to \alpha$ is instantiated with itself to yield $(\Delta\alpha.\ \alpha \to \alpha) \to (\Delta\alpha.\ \alpha \to \alpha)$.

A hierarchy of universes of types can be defined in order to avoid the impredicativity (see, for example, [13]). Here, we will only use two universes: simple types s and polymorphic types τ. The calculus is made predicative by insisting that the quantifier Δ in polymorphic types ranges only over simple types. This can be enforced syntactically by restricting the application of a polymorphic function to simple types.

$$\begin{array}{rrcl}
\text{Simple Types} & s & ::= & \alpha \mid s_1 \to s_2 \\
\text{Polymorphic Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \Delta\alpha.\ \tau \\
\text{Stratified Terms} & M & ::= & x \mid \lambda x{:}\tau.\ M \mid M_1\, M_2 \mid \Lambda\alpha.\ M \mid M\,[s]
\end{array}$$

The inference rules for the principal judgment $\Gamma \vdash M : \tau$ are restricted in the obvious way. We refer to the resulting calculus as the predicative fragment of the polymorphic λ-calculus.

Theorem 26 *Let F be a formula in the unification logic containing only simple types. Then $\lceil F \rceil$ is valid in the polymorphic λ-calculus iff $\lceil F \rceil$ is valid in the predicative fragment of the polymorphic λ-calculus.*

Proof: Examination of the proofs of Lemma 19 and Theorem 21 reveals that all omitted types $[\,]$ in sub-preterms of $\lceil F \rceil$ will be filled by simple types, if all types in the formula F are simple and F is provable. The only explicit type occuring in $\lceil F \rceil$ is α, which is also a simple type. □

Corollary 27 (Predicative Partial Type Reconstruction) *Partial type reconstruction in the predicate fragment of the polymorphic λ-calculus is undecidable.*

Proof: We only need to note that the provability problem for formulas F containing only simple types is undecidable (Theorem 8). □

[3] I am grateful to Robert Harper for this observation.

Acknowledgments

I am grateful to Robert Harper for pointing out that the proof of undecidability does not depend on impredicative features of the polymorphic λ-calculus, to John Greiner for detailed comments on an earlier draft, and to Hans-J. Boehm and Peter Lee for discussions regarding partial polymorphic type reconstruction. This work has been supported in part by the U.S. Air Force under Contract F33615-90-C-1465, ARPA Order No. 7597.

References

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

[2] Hans-J. Boehm. Type inference in the presence of type abstraction. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, Portland, Oregon*, pages 192–206. ACM Press, June 1989.

[3] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM SIGPLAN/SIGACT, 1982.

[4] Paolo Giannini and Simona Ronchi Della Rocca. Type inference in polymorphic type discipline. In T. Ito and A. R. Meyer, editors, *International Conference on Theoretical Aspects of Computer Software, Sendai, Japan*, pages 18–37. Springer-Verlag LNCS 526, September 1991.

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

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

[7] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, 1989.

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

[9] Robert Harper, John Mitchell, and Eugenio Moggi. Higher order modules and the phase distinction. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 341–354. ACM Press, January 1990.

[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] James W. O'Toole Jr. and David K. Gifford. Type reconstruction with first-class polymorphic values. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, Portland, Oregon*, pages 207–217. ACM Press, June 1989.

[12] A. J. Kfoury and J. Tiuryn. Type reconstruction in finite-rank fragments of the polymorphic λ-calculus. *Information and Computation*, 199? To appear.

[13] Daniel Leivant. Finitely stratified polymorphism. *Information and Computation*, 199? To appear. Available as Technical Report CMU-CS-90-160, School of Computer Science, Carnegie Mellon University.

[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 Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 199? To appear.

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

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

[18] John C. Mitchell and Robert Harper. The essence of ML. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 28–46. ACM SIGPLAN/SIGACT, 1988.

[19] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, Snowbird, Utah*, pages 153–163. ACM Press, July 1988.

[20] Frank Pfenning and Peter Lee. LEAP: A language with eval and polymorphism. In *TAPSOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain*, pages 345–359. Springer-Verlag LNCS 352, March 1989.

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