TYPE RECONSTRUCTION IN FINITE-RANK FRAGMENTS OF THE POLYMORPHIC λ-CALCULUS *

(Extended Summary)

A.J. Kfoury
Dept of Computer Science
Boston University

J. Tiuryn†
Institute of Mathematics
University of Warsaw

Abstract

We prove that the problem of type reconstruction in the polymorphic λ-calculus of rank 2 is polynomial-time equivalent to the problem of type reconstruction in ML, and is therefore DEXPTIME-complete. We also prove that for every $k > 2$, the problem of type reconstruction in the polymorphic λ-calculus of rank k, extended with suitably chosen constants with types of rank 1, is undecidable.

1 Introduction

Despite of a lot of activity in the area of type inference (see [1, 3, 4, 7, 13, 14, 15, 16, 18, 21]), the main problem of whether typability is decidable in the full polymorphic λ-calculus of Girard/Reynolds [5, 19], remains open. Instead of attacking it in its full generality, several people have suggested a ramification of the typability problem by "rank". All attempts to solve the problem, even for a small rank ([14, 15]), have been unsuccessful so far.

In this paper we show that for the polymorphic λ-calculus of rank 2, denoted Λ_2, the typability problem is decidable.

This seems to be the first such decidability result for a fragment of the Girard/Reynolds system properly extending the finitely typed λ-calculus. This system, together with the notion of a rank is presented in Section 2. More precisely, we show that typability in Λ_2 is polynomial-time equivalent to typability in **ML**. Thus, by the results of [12] and [8], typability in Λ_2 is DEXPTIME-complete.

We also show that, for every extension of Λ_2 with constants that are typed in rank 1, the typability problem is decidable too. As a corollary, we derive the decidability of the typability problem for an extension of **ML** that consists in allowing a form of "polymorphic abstraction" and any constants of rank 1. We consider such an extension of **ML** mainly to rectify a typing anomaly of the original system, discussed by R. Milner in [17], page 356.

One of the main technical results used in establishing the recursive reducibility of typability in Λ_2 to that in **ML** is the property that, without affecting the power of typability of the system, we can restrict ourselves to instantiations (of bound type variables) with

*Partly supported by NSF grant CCR-8901647 and by a grant of the Polish Ministry of National Education, No. RP.I.09.

†This work was in part carried out while this author was visiting the Computer Science Department of Washington State University, Pullman, WA, during 1988/89, and the Computer Science Department of Boston University, Boston, MA, during the summer of 1989.

CH2897-7/90/0000/0002$01.00 © 1990 IEEE

quantifier-free types. This result is shown in Section 3.

The class of functions numeralwise representable in Λ_2 is strictly larger than the class of functions representable in the finitely typed λ-calculus (e.g., the exponential function is representable in Λ_2). On the other hand, it follows from the results of [20], that the former class of functions is contained in the class of elementary recursive functions. This result is again a consequence of the above mentioned property of Λ_2, namely, restricting Λ_2 to instantiations with quantifier-free types does not affect the power of typability of Λ_2. It is interesting to note, as is shown in [20], that the full polymorphic λ-calculus with instantiations restricted in the way described above, numeralwise represents only elementary recursive functions, while it is well known (see [5]) that the full polymorphic calculus with the unrestricted INST represents all functions provably recursive in second-order arithmetic.

We conclude the paper with the result that for every $k > 2$, the problem of type reconstruction in the polymorphic λ-calculus of rank k, extended with a suitably chosen set C_k of constants with types of rank 1, is undecidable. This system is denoted $\Lambda_k[C_k]$. We reduce the typability problem for $\mathbf{ML}^+$ to the typability problem for $\Lambda_k[C_k]$ ($\mathbf{ML}^+$ in this paper is ML extended with "polymorphic recursion" as defined in [6, 9]). The undecidability of the semi-unification problem (proved in [11]) implies the undecidability of typability in $\mathbf{ML}^+$ (as shown in [10]), which thus implies the undecidability of typability in $\Lambda_k[C_k]$.

2 System Λ_2: The Polymorphic λ-Calculus of Rank 2

We adopt the "Curry view" of the polymorphic λ-calculus, in which pure terms of the λ-calculus are assigned type expressions involving universal quantifiers, rather than the "Church view" where terms and types are defined simultaneously to produce typed terms.

The terms of the pure λ-calculus are defined as usual by the grammar $M ::= x \mid (M\ N) \mid (\lambda x\ M)$. The types we assign to pure λ-terms are defined by the grammar:

$$\tau ::= \alpha \mid (\forall\alpha\ \sigma) \mid (\sigma \to \tau)$$

where α ranges over an infinite set of type variables. We call a type of the form $(\forall\alpha\ \sigma)$ a $\forall$-*type*, and a type of the form $(\sigma \to \tau)$ a *function type*.

We classify types according to the following induction. First define:

$$R(0) \;=\; \{\text{open types}\} = \{\text{types without } \forall\}$$

and then, for all $k \geq 0$, define $R(k+1)$ as the smallest set such that:

$$R(k+1) \;\supseteq\; R(k) \cup \{\ (\forall\alpha\ \sigma) \mid \sigma \in R(k+1)\ \} \cup \{\ (\sigma \to \tau) \mid \sigma \in R(k),\ \tau \in R(k+1)\ \}$$

The set of all types is: $R(\omega) = \bigcup\{R(k) | k \in \omega\}$. We say that $R(k)$ is the set of types of *rank* k. Although defined differently, this hierarchy of types is equivalent to the hierarchy defined earlier (notably in [14] and [15]).

An *assertion* is an expression $A \vdash M : \tau$, with A a type environment (a finite set $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ associating at most one type σ with each variable x), M a term and τ a type — and the *rank* of this assertion is the rank of the type $\sigma_1 \to \cdots \to \sigma_n \to \tau$.

For every $k \geq 0$, we define Λ_k as the fragment of the polymorphic λ-calculus which assigns types of rank $\leq k$ to terms. A precise definition of Λ_k is given in Figure 1; it is the usual type inference system of the polymorphic λ-calculus where all assertions in a derivation are restricted to be of rank k.

We shall use Λ_k to denote both a type inference system and the set of terms typable in that system. Λ_ω is the full polymorphic λ-calculus of Girard/Reynolds.

$\mathrm{FV}(\sigma)$ is the set of free type variables in the type σ and $\mathrm{FV}(A)$ is the set of free type variables in the environment A, defined by:

$$\mathrm{FV}(A) = \bigcup\{\mathrm{FV}(\sigma) | (x : \sigma) \in A\}.$$

The restriction $\alpha \notin \mathrm{FV}(A)$ in the GEN rule in Figure 1 is usually justified on pragmatic grounds (see [2] and [19] for example). The restriction is made in order to avoid "confusion" in the typing of terms. There is a better (technical) justification for the restriction, namely, without this restriction the strong normalization theorem fails since every pure λ-term becomes typable. In fact, without this restriction, every term becomes typable in Λ_2 with a derived type of the form:

$$\forall\alpha_1.(\alpha_1 \to \forall\alpha_2.(\alpha_2 \to \forall\alpha_3.(\ \cdots\ \forall\alpha_{n-1}.(\alpha_{n-1} \to \forall\alpha_n.\alpha_n)\ \cdots\ )))$$

We leave the easy proof of this statement to the reader.

VAR	$A \vdash x : \sigma$	$(x : \sigma) \in A$
INST	$\dfrac{A \vdash M : \forall\alpha.\sigma}{A \vdash M : \sigma[\alpha := \tau]}$	
GEN	$\dfrac{A \vdash M : \sigma}{A \vdash M : \forall\alpha.\sigma}$	$\alpha \notin \mathrm{FV}(A)$
APP	$\dfrac{A \vdash M : \sigma \to \tau, \quad A \vdash N : \sigma}{A \vdash (M\ N) : \tau}$	
ABS	$\dfrac{A[x : \sigma] \vdash M : \tau}{A \vdash (\lambda x\ M) : \sigma \to \tau}$	

Figure 1. System Λ_k: all assertions are of rank k .

3 System Λ_2^-

We introduce another classification of types, denoted $S(0), S(1), S(2), \ldots$. We first define $S(0) = R(0) =$ {open types} and then, for all $k \geq 0$, define $S(k+1)$ as the smallest set such that:

$$S(k+1) \supseteq S(k) \cup \{(\forall\alpha\ \sigma) \mid \sigma \in S(k+1)\} \cup \{\sigma_1 \to \cdots \to \sigma_n \to \tau \mid \sigma_i \in S(k), \tau \in S(0)\}$$

$S(k+1)$ is a proper subset of $R(k+1)$ for all $k \geq 0$. Based on this classification of types[1], we define a new type inference system Λ_2^-, shown in Figure 2.

There are two differences between Λ_2^- and Λ_2. The first is in the type restrictions: in Λ_2^- all environment types are in $S(1)$ and all derived types in $S(2)$. The second difference is between INST and INST$^-$. Observe that the derived type τ in the premise of the ABS rule must not be an $\forall$-type in order to guarantee that the derived type $\sigma \to \tau$ in the conclusion of the same rule be in $S(2)$. The proof of the following is immediate from the definitions.

Lemma 1 *Let M be an arbitrary term. If M is Λ_2^- typable then M is Λ_2 typable.*

We shall show that the converse of Lemma 1 is also true, thus establishing the equivalence of Λ_2^- and Λ_2.

We consider types where some of the quantifiers are marked with #, i.e., types that mention both $\forall$ and $\forall^{\#}$. By a *partially marked type* we mean a type where some (possibly none, possibly all) of the quantifiers are marked with #. By a *totally marked type* we mean a type where all of the quantifiers are marked with #. If σ is any partially marked type, then $(\sigma)^{\#}$ is the totally marked type obtained by marking all quantifiers in σ. The notion of rank introduced in the previous section naturally applies to partially marked types.

The marker # is used to distinguish quantifiers introduced by applications of the INST rule from all other quantifiers. This distinction is made explicit in the system $\Lambda_2^{\#}$ of Figure 3. The essential difference between Λ_2 and $\Lambda_2^{\#}$ is the difference between INST and INST$^{\#}$ (where the notation $\forall^{(\#)}$ means that the quantifier may or may not be marked). The APP rule is changed to APP$^{\#}$ in $\Lambda_2^{\#}$ just to accommodate this difference between INST and INST$^{\#}$.

For partially marked types σ and σ', we write $\sigma \cong \sigma'$ for syntactic equality up to:

1. erasure of all markers,
2. renaming of bound variables (α-conversion),
3. permutation of adjacent quantifiers, i.e., $(\cdots\forall\alpha\forall\beta\cdots) \cong (\cdots\forall\beta\forall\alpha\cdots)$.

Usual equality between (unmarked) type expressions can be taken up to α-conversion and permutation of adjacent quantifiers. The next two lemmas are straightforward consequences of the above definitions.

Lemma 2 *Let M be an arbitrary term. M is Λ_2 typable iff M is $\Lambda_2^{\#}$ typable.*

[1] **Strictly speaking, this is not a real classification of types since not every type is present at some S-level.**

VAR	$A \vdash x : \sigma$	$(x : \sigma) \in A$
INST$^-$	$\dfrac{A \vdash M : \forall\alpha.\sigma}{A \vdash M : \sigma[\alpha := \tau]}$	$\tau \in S(0)$
GEN	$\dfrac{A \vdash M : \sigma}{A \vdash M : \forall\alpha.\sigma}$	$\alpha \notin \mathrm{FV}(A)$
APP	$\dfrac{A \vdash M : \sigma \to \tau, \quad A \vdash N : \sigma}{A \vdash (M\ N) : \tau}$	
ABS	$\dfrac{A[x : \sigma] \vdash M : \tau}{A \vdash (\lambda x\ M) : \sigma \to \tau}$	

Figure 2. System Λ_2^-: all environment types in $S(1)$, all derived types in $S(2)$.

VAR	$A \vdash x : \sigma$	$(x : \sigma) \in A$
INST$^\#$	$\dfrac{A \vdash M : \forall^{(\#)}\alpha.\sigma}{A \vdash M : \sigma[\alpha := (\tau)^\#]}$	
GEN	$\dfrac{A \vdash M : \sigma}{A \vdash M : \forall\alpha.\sigma}$	$\alpha \notin \mathrm{FV}(A)$
APP$^\#$	$\dfrac{A \vdash M : \sigma \to \tau, \quad A \vdash N : \sigma'}{A \vdash (M\ N) : \tau}$	$\sigma \cong \sigma'$
ABS	$\dfrac{A[x : \sigma] \vdash M : \tau}{A \vdash (\lambda x\ M) : \sigma \to \tau}$	

Figure 3. System $\Lambda_2^\#$: all assertions are of rank 2, all environment types are unmarked (#-free).

Lemma 3 *Let σ be a partially marked type. If σ is a derived type in $\Lambda_2^\#$ (i.e., there is a term M and an environment A such that $A \vdash M : \sigma$ in $\Lambda_2^\#$), then no unmarked quantifier in σ is within the scope of a marked quantifier:*

$$\sigma \neq (\cdots(\forall^\#\alpha.\cdots(\forall\beta.\cdots)\cdots)\cdots)$$

for any type variables α and β.

We require throughout that in every type σ the bound variables are disjoint from the free variables, $BV(\sigma) \cap FV(\sigma) = \emptyset$, and no variable is bound more than once. This requirement is satisfied by α-conversion.

We define a mapping that assigns to every partially marked type σ, an unmarked type $\sigma^\bullet$.

1. $\alpha^\bullet = \alpha$,
   where α is a type variable,
2. $(\sigma \to \tau)^\bullet = \forall\vec{\alpha}.(\sigma^\bullet \to \rho)$,
   where $\tau^\bullet = \forall\vec{\alpha}.\rho$ and ρ is not a $\forall$-type,
3. $(\forall\alpha.\sigma)^\bullet = \forall\alpha.\sigma^\bullet$,
4. $(\forall^\#\alpha.\sigma)^\bullet = \sigma^\bullet$

In 2 above, $\vec{\alpha}$ denotes a finite (possibly empty) set of type variables $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, and $\forall\vec{\alpha}$ denotes $\forall\alpha_1\forall\alpha_2\cdots\forall\alpha_n$. The map $(\ )^\bullet$ accomplishes two things – first, it displaces unmarked quantifiers leftward (as much as possible without changing the "meaning" of types) and, second, it eliminates all marked quantifiers.

Lemma 4 *For all $k \geq 0$ and for every partially marked type σ, if $\sigma \in R(k)$ then $\sigma^\bullet \in S(k)$ and $FV(\sigma) \subseteq FV(\sigma^\bullet)$.*

Proof idea: By induction on $k \geq 0$. It suffices to prove the lemma for types σ without marked quantifiers, in which case we also prove that $FV(\sigma) = FV(\sigma^\bullet)$. ∎

Lemma 5 *Let σ_1 and σ_2 be derived types in $\Lambda_2^\#$ such that $\sigma_1 \cong \sigma_2$ and $\sigma_1, \sigma_2 \in R(1)$. If $\sigma_1^\bullet = \forall\vec{\beta}_1.\pi_1$ and $\sigma_2^\bullet = \forall\vec{\beta}_2.\pi_2$, where π_1 and π_2 are not $\forall$-types, then we can rename bound variables in σ_1 and σ_2 (α-conversion) so that: (1) $\pi_1 = \pi_2$ and (2) $\vec{\beta}_1 \subseteq \vec{\beta}_2$ or $\vec{\beta}_2 \subseteq \vec{\beta}_1$.*

Proof: This follows from the definition of $(\ )^\bullet$ and Lemma 3. ∎

Lemma 6 *Let σ a partially marked type, τ a totally marked type, and α a type variable. Then $\sigma^\bullet[\alpha := \tau^\bullet] = (\sigma[\alpha := \tau])^\bullet$.*

Proof: This follows from the definition of $(\ )^\bullet$ (the hypothesis that τ is *totally* marked is essential). ∎

If A is a type environment then

$$A^\bullet = \{(x : \sigma^\bullet) | (x : \sigma) \in A\}$$

Every type σ in A is unmarked, so that $FV(\sigma) = FV(\sigma^\bullet)$, and therefore $FV(A) = FV(A^\bullet)$.

Lemma 7 *Let M be an arbitrary term. If M is $\Lambda_2^\#$ typable then M is Λ_2^- typable; more specifically, for every (partially marked) type σ and environment A, if $A \vdash M : \sigma$ in $\Lambda_2^\#$ then $A^\bullet \vdash M : \sigma^\bullet$ in Λ_2^-.*

Proof idea: By induction on the length ≥ 1 of derivations in $\Lambda_2^\#$, using Lemma 4 ("the rank of a derivation remains the same when passing from $\Lambda_2^\#$ to Λ_2^-"), Lemma 5 and Lemma 6. ∎

Theorem 8 *Let M be an arbitrary term. M is Λ_2^- typable iff M is Λ_2 typable.*

Proof: The left-to-right implication is Lemma 1. The right-to-left implication follows from Lemmas 2 and 7. ∎

4 Typability in the Polymorphic λ-Calculus of Rank 2

In this section we show that Λ_2 typability and **ML** typability are polynomial-time reducible to each other. *Terms of* **ML** are defined according to the following syntax.

$$M ::= x | (MN) | (\lambda x.M) | (\textbf{let } x = N \textbf{ in } M)$$

Thus pure terms form a proper subset of **ML** terms. Throughout this section we assume that terms M satisfy the following two restrictions.

1. no variable is bound more than once in M,
2. bound and free variables in M are disjoint.

These assumptions can be easily met by α-conversion. In Figure 4 we state the typing system for **ML** terms (cf. [1]).

VAR	$A \vdash x : \sigma$	$(x : \sigma) \in A$
INST	$\dfrac{A \vdash M : \forall\alpha.\sigma}{A \vdash M : \sigma[\alpha := \tau]}$	
GEN	$\dfrac{A \vdash M : \sigma}{A \vdash M : \forall\alpha.\sigma}$	$\alpha \notin \mathrm{FV}(A)$
APP	$\dfrac{A \vdash M : \tau \to \tau', \quad A \vdash N : \tau}{A \vdash (M\ N) : \tau'}$	
ABS	$\dfrac{A[x : \tau] \vdash M : \tau'}{A \vdash (\lambda x\ M) : \tau \to \tau'}$	
LET	$\dfrac{A[x : \sigma] \vdash M : \tau \quad A \vdash N : \sigma}{A \vdash (\mathbf{let}\ x = N\ \mathbf{in}\ M) : \tau}$	

Figure 4. System **ML**, $\sigma \in S(1)$, $\tau, \tau' \in S(0)$

We will also consider $\mathbf{ML_1}$, an extension of **ML** which allows polymorphic abstraction. We consider such an extension of **ML** as a response to the critique of the restriction[2] in the original system that forces all occurrences of the same λ-bound variable to have the same type. $\mathbf{ML_1}$ is obtained from **ML** by exchanging the APP and ABS rules with the rules of Figure 5.

We refer to $\mathbf{ML_1}$ as **ML** with *polymorphic abstraction*. The original system **ML** contains other programming constructs, such as **if-then-else** or **let-rec** (monomorphic recursion). Clearly they can be reintroduced into our presentation via suitably typed constants.

Lemma 9 *If $A \vdash M : \sigma$ is derivable in* **ML** *(or in Λ_2^-), then there is a derivation of $A \vdash M : \sigma$ in* **ML** *(or in Λ_2^-, respectively) in which INST rule is applied only to variables, i.e., instead of* INST *the following more restrictive rule* $\mathrm{INST}^{\mathbf{var}}$ *is used throughout the derivation*

$$\mathrm{INST}^{\mathbf{var}} \qquad \frac{A \vdash x : \forall\alpha.\sigma}{A \vdash x : \sigma[\alpha := \tau]} \qquad (\tau \in S(0))$$

Proof idea: By induction on M. A substitution that corresponds to an application of INST to a term that is not a variable can be easily "pushed" all the way towards the leaves of the derivation tree. ∎

[2]Identified as "the main limitation of the system" in R. Milner's original paper [17], page 356.

Let us define, by induction on **ML** terms M, the sequence $act(M)$, of *active variables* in M.

1. $act(x) = \varepsilon$ (the empty sequence)
2. $act(\lambda x.M) = act(M) * x$
3. $act(MN) =$
$$\begin{cases} \varepsilon & \text{if } act(M) = \varepsilon \\ x_1 \cdots x_{n-1} & \text{if } act(M) = x_1 \cdots x_n \text{ and } n \geq 1 \end{cases}$$
4. $act(\mathbf{let}\ x = N\ \mathbf{in}\ M) = act(M)$

Let x be a variable and let M be an **ML** term. We define M_x, the effect of deleting λx in M, by induction on M as follows:

1. $(y)_x = y$, (x, y are variables)
2. $(\lambda y.M)_x = \begin{cases} M & \text{if } y = x \\ \lambda y.M_x & \text{if } y \neq x \end{cases}$
3. $(MN)_x = (M_x)(N_x)$
4. $(\mathbf{let}\ y = N\ \mathbf{in}\ M)_x = (\mathbf{let}\ y = N_x\ \mathbf{in}\ M_x)$

Lemma 10 *Let M be an* **ML** *term such that $act(M) \neq \varepsilon$ and let x be the rightmost variable in $act(M)$. Then for all types $\tau, \rho \in S(0)$ and for every environment A:*

$$A[x : \tau] \vdash_{\mathbf{ML}} M_x : \rho \qquad iff \qquad A \vdash_{\mathbf{ML}} M : \tau \to \rho$$

$$\text{APP}^+ \qquad \frac{A \vdash M : \sigma \to \tau, \qquad A \vdash N : \sigma}{A \vdash (M\ N) : \tau}$$

$$\text{ABS}^+ \qquad \frac{A[x : \sigma] \vdash M : \tau}{A \vdash (\lambda x\ M) : \sigma \to \tau}$$

Figure 5. New rules for $\mathbf{ML}_1$, $\sigma \in S(1)$, $\tau \in S(0)$

Proof idea: By induction on M, using Lemma 9. ■

Finally, we define a mapping that assigns to every pure term M an **ML** term $\hat{M}$.

1. $\hat{x} = x$
2. $\widehat{\lambda x.M} = \lambda x.\hat{M}$
3. $\widehat{MN} =$
$$\begin{cases} \hat{M}\hat{N} & \text{if } act(M) = \varepsilon \\ \textbf{let } y = \hat{N} \textbf{ in } (\hat{M})_y & \text{if } act(M) = x \cdots y \end{cases}$$

Lemma 11 *Let M be a pure term such that $act(M) = x_1 \cdots x_n$ for some $n \geq 0$. If $A \vdash_{\Lambda_2^-} M : \sigma$, for some environment A and $\sigma \in S(2)$, then*

$$\sigma = \forall \vec{\alpha}.(\sigma_n \to \cdots \to \sigma_1 \to \tau)$$

where $\sigma_1, \ldots, \sigma_n \in S(1)$, $\tau \in S(0)$, and $\vec{\alpha}$ is a sequence of type variables (possibly empty).

Proof: Routine induction on the length of a derivation. ■

Lemma 12 *For every pure term M such that $act(M) = x_1 \cdots x_n$, for some $n \geq 0$, and for every environment A and type $\sigma = \sigma_n \to \cdots \to \sigma_1 \to \tau$, where $\sigma_1, \ldots, \sigma_n \in S(1)$ and $\tau \in S(0)$:*

$$A \vdash_{\Lambda_2^-} M : \sigma \qquad iff$$

$$A[x_1 : \sigma_1, \ldots, x_n : \sigma_n] \vdash_{\text{ML}} (\hat{M})_{x_1 \cdots x_n} : \tau$$

Proof idea: By induction on pure terms M, using Lemmas 9, 10 and 11. ■

Combining Lemma 11, Lemma 12 and Theorem 8 we obtain the following result.

Theorem 13 *For every pure term M, if $act(M) = x_1 \cdots x_n$, for some $n \geq 0$, then:*

M is typable in Λ_2 iff $(\hat{M})_{x_1 \cdots x_n}$ is typable in **ML**

Thus, Λ_2 typability is polynomial-time reducible to **ML** *typability.*

We next establish a polynomial-time reduction in the opposite direction. Let us define a mapping that assigns to every **ML** term M a pure term $M^\sim$. Let $I = (\lambda x.x)$ be the identity combinator.

1. $x^\sim = x$
2. $(\lambda x.M)^\sim = I(\lambda x.M^\sim)$
3. $(MN)^\sim = M^\sim N^\sim$
4. $(\textbf{let } x = N \textbf{ in } M)^\sim = (\lambda x.M^\sim)N^\sim$

Theorem 14 *For every* **ML** *term M, environment A, and type $\sigma \in S(1)$:*

$$A \vdash_{\text{ML}} M : \sigma \qquad iff \qquad A \vdash_{\Lambda_2^-} M^\sim : \sigma$$

Thus, by Theorem 8, **ML** *typability is polynomial-time reducible to Λ_2 typability.*

Proof: By induction on **ML** terms M. The left-to-right implication is a routine induction (omitted here). For the right-to-left implication we prove a somewhat stronger statement.

Claim: *For every* **ML** *term M, environment A, and type $\sigma \in S(2)$, if $A \vdash_{\Lambda_2^-} M^\sim : \sigma$, then $\sigma \in S(1)$ and $A \vdash_{\text{ML}} M : \sigma$.*

The proof of the claim is by induction on **ML** terms M, using Lemma 9, omitted here. ■

Theorem 15 *Given an environment A that assigns types of rank 1 and a pure term M, it is decidable whether there exists an environment B, a substitution S, and a type $\sigma \in R(2)$ such that $A \subseteq B$ and $S(B) \vdash_{\Lambda_2} M : \sigma$.*

Proof: For $\mathcal{K}$ being one of the systems Λ_2, Λ_2^-, or **ML**, let $P_{\mathcal{K}}(A, M)$ be the property of an environment A and a term M that expresses the existence of an environment B, a substitution S, and a type σ (of an appropriate rank) such that $A \subseteq B$ and $S(B) \vdash_{\mathcal{K}} M : \sigma$. Let A be any environment that assigns only types of rank 1 and let M be any pure term. Then, by Lemma 7, $P_{\Lambda_2}(A, M)$ iff $P_{\Lambda_2^-}(A^\bullet, M)$. By Lemmas 11 and 12, $P_{\Lambda_2^-}(A^\bullet, M)$ iff $P_{\mathrm{ML}}(A^\bullet, (\lambda M)_{x_1 \cdots x_n})$, where $act(M) = x_1 \cdots x_n$, $n \geq 0$. The desired conclusion follows from the decidability of P_{ML}, see [1]. ∎

Next we will consider an extension of the system Λ_k by constants. A *typing of constants* is any finite function C from the set of constant symbols to the set of closed types. Let C be a typing of constants. Pure terms with constants in C are defined by the following grammar:

$$M ::= x | c | (MN) | (\lambda x.M)$$

where c ranges over the constant symbols in the domain of C.

$\Lambda_k[C]$ is the extension of the system Λ_k (see Fig.1) by the following rule:

$$\mathrm{CONST}^C \qquad A \vdash c : \sigma \qquad (C(c) = \sigma)$$

Clearly for the above rule to conform to the general restrictions of types in derivations of Λ_k we must request that the typing of constants C assigns types of rank k to all constants in the domain. We say that the typing of constants *is of rank* k if it assigns only types of rank k to all constants in the domain. We omit the straightforward proofs of the next two corollaries.

Corollary 16 *For every typing C of constants of rank 1, typability in $\Lambda_2[C]$ is decidable.*

Corollary 17 *Let C be a typing of constants that assigns to each constant in its domain a type in $S(1)$. Then typability in $\mathbf{ML}_1[C]$, **ML** extended by polymorphic abstraction and constants in C, is decidable.*

5 Typability in the Polymorphic λ-Calculus of Higher Ranks

We will need the following definition. Let X be a set of type variables and let σ, τ be two types. τ is an *X-instance* of σ, denoted $\sigma \leq_X \tau$, if there is a substitution S and a type τ' such that $S(\sigma) = \tau'$ (here S obviously acts only on the free variables of σ), $dom(S) \cap X = \emptyset$ and $\tau = \forall\alpha_1 \ldots \forall\alpha_n.\tau'$, where $\alpha_1, \ldots, \alpha_n \notin X$.

Let us define a sequence of open types $\tau_1, \tau_2, \ldots$ as follows. Let $\tau_1 = (\alpha \to \alpha)$ and let $\tau_{k+1} = (\tau_k \to \alpha)$. Let $k \geq 3$, and let C_k be the following typing of constants.

$$c : \forall\alpha.(\alpha \to \tau_k)$$

$$f : \forall\alpha.((\alpha \to \alpha) \to \tau_{k-1})$$

In this section we show the following result.

Theorem 18 *For every $k \geq 3$, typability in $\Lambda_k[C_k]$ is undecidable.*

We prove Theorem 18 by showing that typability in an appropriate extension of **ML**, denoted $\mathbf{ML}^+$, is polynomial time reducible to typability in $\Lambda_k[C_k]$. It follows from the results of [10] (our $\mathbf{ML}^+$ here and in [9] is denoted $\mathbf{ML}/1$ in [10]), and from the undecidability of the semi-unification problem (see [11]) that typability in $\mathbf{ML}^+$ is undecidable.

First we recall the system $\mathbf{ML}^+$. It differs from **ML** by allowing a richer rule for typing recursion. The terms of **ML** defined in the previous section do not contain recursion construct since it was not necessary for the reducibility result of the previous section. For the purposes of this paper we define *terms* of $\mathbf{ML}^+$ as follows.

$$M ::= x | (MN) | (\lambda x.M) | (\mathbf{fix} x.M)$$

The typing rules for $\mathbf{ML}^+$ are the rules VAR, INST, GEN, APP, ABS of **ML** plus the following rule for recursion:

$$\mathrm{FIX}^+ \qquad \frac{A[x : \sigma] \vdash M : \sigma}{A \vdash \mathbf{fix} x.M : \sigma} \qquad (\sigma \in S(1))$$

To establish the reducibility result of this section let us define a mapping that assigns to every $\mathbf{ML}^+$ term M a pure term $M^\flat$ with constants in C_k.

1. $x^\flat = cxy$
2. $(MN)^\flat = (cM^\flat y)(cN^\flat z)$
3. $(\lambda x.M)^\flat = c(\lambda x.M^\flat)y$
4. $(\mathbf{fix} x.M)^\flat = f(\lambda x.M^\flat)y$

In the above definition we assume that the new variables y, z, introduced in $M^\flat$ do not occur in M and that they are pairwise different.

To control quantifiers introduced by INST rule of $\Lambda_k[C_k]$ we use the system $\Lambda_k^{\#}[C_k]$ obtained from Λ_k in essentially the same way the system $\Lambda_2^{\#}$ was obtained from Λ_2 at the beginning of Section 3 (see Fig.3). The essential feature of $\Lambda_k^{\#}[C_k]$ is that every quantifier introduced by INST rule is marked with $\#$. Then the mapping that assigns to a marked type σ an unmarked type $\sigma^\bullet$ erases all marked quantifiers and moves all other quantifiers to the front as far as possible (see the definition in Section 3). Clearly both systems $\Lambda_k[C_k]$ and $\Lambda_k^{\#}[C_k]$ are equivalent with respect to typability.

Lemma 19 is immediate from the definitions. The proofs of Lemmas 20 and 21 are not too difficult, and omitted for lack of space.

Lemma 19 *For every set X of type variables,*

(i) *$\leq_X$ is transitive, and*

(ii) *for all marked types σ and τ, if $\sigma \leq_X \tau$ then $\sigma^\bullet \leq_X \tau^\bullet$.*

Lemma 20 *Let M be a pure term with constants in C. If $A \vdash_{\Lambda_k[C_k]} cMy : \sigma$, then there exists $\tau \in R(0)$ such that $\tau \leq_{FV(A)} \sigma$, and $A \vdash_{\Lambda_k[C_k]} M : \tau$.*

Lemma 21 *If $A \vdash_{\Lambda_k[C_k]} fMy : \sigma$, then there is $\tau \in R(1)$ such that $\tau \leq_{FV(A)} \sigma$ and $A \vdash_{\Lambda_k[C_k]} M : \tau \to \tau$.*

Now we are ready to state the main lemma in the proof of the reducibility result.

Lemma 22 *Let A be an environment and let M be an $\mathbf{ML}^+$ term, such that for every variable x, if $x \notin FV(M^\flat)$ and if $(x : \rho) \in A$, then $\rho \in R(1)$. Then, for every type σ, if $A \vdash_{\Lambda_k^{\#}[C_k]} M^\flat : \sigma$, then $A^\bullet \vdash_{\mathbf{ML}^+} M : \sigma^\bullet$.*

Proof idea: By induction on M, using the three preceding lemmas. ■

Lemma 23 *For every environment A, $\mathbf{ML}^+$ term M, and a type σ, if $A \vdash_{\mathbf{ML}^+} M : \sigma$, then there is an environment $B \supseteq A$ such that $B \vdash_{\Lambda_k[C_k]} M^\flat : \sigma$.*

Proof: Obvious induction on M. ■

By Lemma 22 and Lemma 23 we can immediately establish an effective reduction of typability in $\mathbf{ML}^+$ to typability in $\Lambda_k[C_k]$.

Lemma 24 *Let $k \geq 3$. For every term M of $\mathbf{ML}^+$, M is typable in $\mathbf{ML}^+$ iff $M^\flat$ is typable in $\Lambda_k[C_k]$.*

Acknowledgement We are grateful to Paweł Urzyczyn for his comments and stimulating discussions on the material contained in this paper.

References

[1] Damas, L. and Milner, R., "Principal type schemes for functional programs," *Proc. 9-th ACM Symp. Principles of Prog. Lang.*, pp 207-212, 1982.

[2] Fortune, S., Leivant, D., and O'Donnell, M., "The expressiveness of simple and second-order type structures", *J. ACM*, **30**, pp 151-185, 1983.

[3] Giannini, P., "Type-checking and type deduction techniques for polymorphic programming languages", Technical Report, Dept of Computer Science, CMU, Dec 1985.

[4] Giannini, P., Ronchi Della Rocca, S., "Characterization of typings in polymorphic type discipline", *Proc of IEEE 3-rd LICS*, July 1988.

[5] Girard, J.-Y., *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieure*, Doctoral thesis, Université Paris VII, 1972.

[6] Henglein, F., "Type inference and semi-unification", *Proc. ACM Symp. LISP and Functional Programming*, July 1988.

[7] Hindley, J.R., "The principal type-scheme of an object in combinatory logic", *Transactions of the American Mathematical Society*, **146**, pp 29-60, Dec 1969.

[8] Kanellakis, P., Mairson, H.G. and Mitchell, J.C., "Unification and ML typing", submitted for publication. Preliminary versions appeared in *POPL 1989* and *POPL 1990*, under the respective titles "Polymorphic unification and ML typing" (by Kanellakis and Mitchell) and "Deciding ML typability is complete for deterministic exponential time" (by Mairson).

[9] Kfoury, A.J., Tiuryn, J. and Urzyczyn, P., "A proper extension of ML with an effective type-assignment," *Proc. 15-th ACM Symp. Principles of Programming Languages*, 1988.

[10] Kfoury, A.J., Tiuryn, J. and Urzyczyn, P., "Type-checking in the presence of polymorphic recursion", Boston University Research Report, 1989. Part of the results of this paper were presented in *Proc. of IEEE 4-th LICS, 1989*, under the title "Computational consequences and partial solutions of a generalized unification problem".

[11] Kfoury, A.J., Tiuryn, J. and Urzyczyn, P., "The undecidability of the semi-unification problem", Boston University Research Report, No. 89-011, Oct 1989.

[12] Kfoury, A.J., Tiuryn, J. and Urzyczyn, P., "An analysis of ML typability", Boston University Research Report, No. 89-009, Oct 1989.

[13] Leiss, H., "On type inference for object-oriented programming languages", *Proc. Logik in der Informatik*, Karlsruhe, Springer-Verlag, Dec 1987.

[14] Leivant, D., "Polymorphic type inference", *Proc. of 10-th POPL*, ACM, 1983.

[15] McCracken, N., "The typechecking of programs with implicit type structure", in *Semantics of Data Types*, ed. by Kahn, McQueen and Plotkin, Springer-Verlag LNCS 173, 1984.

[16] Mitchell, J.C., "Polymorphic type inference and containment", *Information and Computation*, **76**, 2/3, 1988.

[17] Milner, R., "A theory of type polymorphism in programming", ***J. of Computer and System Sciences***, Vol. 17, pp 348-375, 1978.

[18] Pfenning, F., "Partial polymorphic type inference and higher-order unification", *Proc. of Lisp and Functional Programming Conference*, 1988.

[19] Reynolds, J., "Towards a theory of type structure", *Proc. Colloque sur la Programmation*, pp 408-425, Springer-Verlag LNCS 19, 1974.

[20] Tiuryn, J., "Representability of arithmetic functions in fragments of second-order λ-calculus", Manuscript, 1988.

[21] Wand, M., "Complete type inference for simple objects", *Proc. 2nd IEEE Symposium on Logic in Computer Science*, pp 37-44, 1987. See also "Corrigendum: complete type inference for simple types", *Proc. 3rd IEEE Symposium on Logic in Computer Science*, p 132, 1988.