INFORMATION AND COMPUTATION 98, 228–257 (1992)

Type Reconstruction in Finite Rank Fragments of the Second-Order λ-Calculus*

A. J. KFOURY

Department of Computer Science, Boston University, Boston, Massachusetts 02215

AND

J. TIURYN†

Institute of Informatics, Warsaw University, 01-913 Warszawa, ul. Banacha 2, Poland

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.

Contents

1. *Introduction.*
2. *System Λ_k: The polymorphic λ-calculus of rank k.*
3. *System Λ_2^-.*
4. *Typability in the polymorphic λ-calculus of rank 2.*
5. *Typability in the polymorphic λ-calculus of higher ranks.*
6. *Concluding remarks.*

1. INTRODUCTION

Despite considerable activity in the area of type inference (see Damas and Milner, 1979; Giannini, 1985; Giannini and Ronchi Della Rocca, 1988;

* This work is partly supported by NSF Grant CCR-8901647 and by a grant of the Polish Ministry of National Education, No. RP.I.09. A preliminary version of this paper appeared in the *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, pp. 2–11, IEEE, New York, 1990.

† This work was in part carried out while this author was visiting the Computer Science Department of Washington State University, Pullman, Washington, during the academic year 1988–1989, and the Computer Science Department of Boston University, Boston, Massachusetts, during the summer of 1989.

0890-5401/92 $5.00

Mindley, 1969; Leiss, 1987; Leivant, 1983; McCracken, 1984; Mitchell, 1988; Pfenning, 1988; and Wand, 1987), the main problem of whether typability in the full polymorphic λ-calculus F_2 of Girard (1972) and Reynolds (1974) is decidable, remains open. A recent result of Henglein and Mairson (1991) shows that this problem is DEXPTIME-hard.

Instead of attacking the main problem in its full generality, several people have suggested a ramification of the problem by "rank." In this paper we consider the notion of rank which is equivalent to the notion of rank introduced by Leivant (1983). Another stratified system for F_2 was recently introduced in Giannini and Ronchi Della Rocca (1991). The two systems are incomparable in the sense that strata of one are not in general fully contained in strata of the other. Leivant (1983) considers the problem of type reconstruction for F_2, as well as for the conjunctive type discipline of Coppo and Dezani-Ciancaglini (1979, 1981). Leivant sketches a proof of decidability of type reconstruction for conjunctive types whose rank is restricted to 2. McCracken (1984) presents an algorithm which is based on the ideas of Leivant (1983) for deciding typability in the polymorphic λ-calculus of rank 2. However, none of those algorithms was proven correct[1] and full versions of the proceedings papers (Leivant, 1983; McCracken, 1984) have never been published.

In this paper we show, using a different method of proof than in (Leivant, 1983; McCracken, 1984) that for the polymorphic λ-calculus of rank 2, denoted Λ_2, the typability problem is decidable. 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 the typability in **ML**. Thus, by the results of (Kanellakis, Mairson, and Mitchell, to appear, and Kfoury, Tiuryn, and Urzyczyn, to appear), typability in Λ_2 is DEXPTIME-complete.

We also show that, for every extension of Λ_2 with constants that are assigned universally polymorphic types, 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 assigned universally polymorphic types. We consider such an extension of **ML** mainly to rectify a typing anomaly of the original system, as discussed by Milner (1978, p. 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

[1] The algorithm D in McCracken (1984, p. 311), as stated in the paper, is clearly incorrect. For example, procedure "alphavary" in the 3rd line of the algorithm should be applied to the environment A and not only to the type of x.

instantiations (of bound type variables) with 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 result of Leivant (1989) and Tiuryn (1988) 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.

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 $\Lambda_k[C_k]$ ($\mathbf{ML}^+$ is **ML** extended with "polymorphic recursion" as defined in Henglein (1988), Kfoury, Tiuryn, and Urzyczyn (1988), and Mycroft (1984)). The undecidability of the semi-unification problem (proved in Kfoury, Tiuryn, and Urzyczyn, to appear) implies the undecidability of typability in $\mathbf{ML}^+$, which thus implies the undecidability of typability in $\Lambda_k[C_k]$.

In order to avoid a possible confusion we briefly discuss what we believe are three different approaches to type reconstruction problems. Suppose we are given a system $\vdash$ for deriving assertions of the form $A \vdash M: \sigma$, where A is an environment which assigns types to object variables, M is a term of the λ-calculus, and σ is a type.

The first and most popular type reconstruction problem is: given a term M, decide whether there exist A and σ such that $A \vdash M: \sigma$ is derivable. We refer to this problem as the *type reconstruction problem* for $\vdash$.

The second problem, which we call the *strong type reconstruction problem* for $\vdash$ (see Tiuryn, 1990, where the name was introduced) is: given A and M, is there B and σ such that B extends A (i.e., B does not change types assigned to variables by A—it is perhaps more appropriate to call "constants" these "variables" with pre-assigned types) and $B \vdash M: \sigma$ is derivable. Hence, strong type reconstruction problem represents the situation of built-in constants which come with a typing as a part of the design of the language. Typical of such constants are **if–then–else** and the fixed-point operator.

The third and the most general problem is the problem of type reconstruction for partially typed terms. The freedom with which we can constrain relationships between types is much greater here than in the previous two cases. A partially typed term (see Boehm, 1985) has some or all of the following features: constants are typed (as in the case of strong type reconstruction); some object abstractions may be equipped with types

and some may be untyped; type application either is explicit (i.e., a term is applied to an explicit type) or it is implicit (i.e., it is marked by an application of a term to a "place-holder") type abstraction is obligatory (i.e., the reconstruction of missing type information cannot introduce new type abstractions). In order to state the problem of type reconstruction for partially typed terms we need a type derivation system $\vdash$ for terms which are fully typed, i.e., terms in which every object abstraction is typed and all type applications and type abstractions are explicit. Then the problem is formulated as follows: given a partially typed term M and an environment A (a typing of constants), can M be completed to $\tilde{M}$ by inserting the missing type information, so that for some environment B which extends A and for some type σ, $B \vdash \tilde{M}: \sigma$ is derivable.

The ordinary problem of type reconstruction is obviously a special case of the strong type reconstruction problem (with the empty environment). Hence undecidability of the former implies undecidability of the latter but not necessarily the other way round.[2] Because of the obligatory type abstraction and (at least implicit) obligatory type application there is no clear relationship between the first two problems and the problem for partially typed terms. The latter problem was shown to be undecidable by Boehm (1985) for the system F_2 with explicit typing.

2. System Λ_k: The Polymorphic λ-Calculus of Rank k

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 then "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 use the standard convention according to which arrows associate to the right, i.e., $\sigma_1 \to \cdots \to \sigma_n$ is an abbreviation for $(\sigma_1 \to \cdots \to (\sigma_{n-2} \to (\sigma_{n-1} \to \sigma_n)) \cdots)$. Types which differ only by names of bound variables (bound by $\forall$) are considered equal (α-conversion).

[2] However, we do not have any natural example supporting this remark.

We classify types according to the following induction. First define

$$R(0) = \{\text{open types}\} = \{\text{types not mentioning } \forall\}$$

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

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

The set of all types is: $R(\omega) = \bigcup \{R(k) \mid k \in \omega\}$. $R(k)$ is the set of types of rank k. For example, $\forall\alpha\ (\alpha \to \forall\beta\ (\alpha \to \beta))$ is a type of rank 1 and $\forall\alpha\ (\alpha \to \alpha) \to \forall\beta\ \beta$ is a type of rank 2 but not of rank 1. It is easy to prove inductively that a type σ has rank k iff there is no instance of $\forall$ falling in σ in the negative scope (i.e., left hand side) of k nested instances of $\to$. Hence our definition of rank is equivalent to the notion of rank introduced in Leivant (1983). It also follows from this observation that every type is assigned a rank; i.e., $R(\omega)$ is the set of all types. Since $R(k) \subseteq R(k+1)$ it follows that if a type σ has a rank k, then it has every rank $n \geqslant k$.

An *assertion* is an expression of the form $A \vdash M : \tau$, where A is a type environment (a finite set $\{x_1 : \sigma_1, ..., x_n : \sigma_n\}$ 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 $\sigma_1 \to \cdots \to \sigma_n \to \tau$. In particular an assertion $A \vdash M : \tau$ is of rank 2 iff τ is of rank 2 and all the environment types are of rank 1. $A[x : \sigma]$ is the environment obtained from A by removing a pair $(x : \tau)$ from A (if there is one) and adding $(x : \sigma)$.

For every $k \geqslant 0$, we define Λ_k as the fragment of the polymorphic λ-calculus which is a restriction of F_2 to assertions of rank k. A precise definition of Λ_k is given in Fig 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.

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	$\dfrac{A[x : \sigma] \vdash M : \tau}{A \vdash (\lambda x\ M) : \sigma \to \tau}$	

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

We use Λ_k to denote both a type inference system (as a set of rules for deriving assertions) and the set of terms typable in that system (i.e., the terms M such that for some A and σ, $A \vdash M : \sigma$ is derivable in Λ_k). $\bigcup \{\Lambda_k \mid k \geq 0\} = F_2$ is the full polymorphic λ-calculus. If $\mathscr{K}$ is any type inference system, then we will be using the notation $A \vdash_{\mathscr{K}} M : \tau$ to denote that $A \vdash M : \tau$ is derivable in $\mathscr{K}$.

$FV(\sigma)$ is the set of free type variables in the type σ (i.e., variables which are not bound by a quantifier) and $FV(A)$ is the set of free type variables in the environment A, defined by $FV(A) = \bigcup \{FV(\sigma) \mid (x : \sigma) \in A\}$. By $\sigma[\alpha := \tau]$ we denote the result of substituting τ for all free occurrences of α in σ (a renaming of bound variables may be necessary before the substitution is performed in order to avoid clashes of variables). In general, performing the substitution $\sigma[\alpha := \tau]$ may increase the rank of σ. The resulting rank depends on the rank of τ and how deep in the negative scope of $\rightarrow$ are free occurrences of α in σ.

3. System Λ_2^-

In this section we introduce a type inference system Λ_2^- (see Fig. 2) which is based on a more restrictive set of types than the system Λ_2. The main result of this section shows that these two systems are equivalent with respect to typable terms. Because Λ_2^- is more restrictive, it is easier to analyze the problem of type reconstruction.

In order to define Λ_2^- we have to restrict types of rank 2 to a special syntactic form. Let $S(0)$ be the set of all open types; let $S(1)$ be the set of all types of the form $\forall\alpha_1 \cdots \forall\alpha_n \sigma$, where $\sigma \in S(0)$. Let $S(2)$ be the set of all

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	$\dfrac{A \vdash M : \sigma \rightarrow \tau, \quad A \vdash N : \sigma}{A \vdash (M\ N) : \tau}$	
ABS	$\dfrac{A[x : \sigma] \vdash M : \tau}{A \vdash (\lambda x\ M) : \sigma \rightarrow \tau}$	

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

types of the form $\forall\alpha_1 \cdots \forall\alpha_n\ (\sigma_1 \to \cdots \to \sigma_m \to \tau)$, where $\sigma_1, ..., \sigma_m \in S(1)$ and $\tau \in S(0)$.

Let us observe that $S(1)$ is a proper subset of types of rank 1 and $S(2)$ is a proper subset of types of rank 2. Types in $S(1)$ are what is called in **ML** *type schemes*. Types in $S(2)$ form a minimal class for which it is possible to have in the language procedures which pass univerally polymorphic (i.e., of type in $S(1)$) parameters.

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 a $\forall$-type in order to guarantee that the derived type $\sigma \to \tau$ in the conclusion of the same rule be in $S(2)$.

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

Proof. For all A and σ, if $A \vdash_{\Lambda_2^-} M : \sigma$ then $A \vdash_{\Lambda_2} M : \sigma$. ∎

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

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 (possible none, possibly 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 Fig. 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 erasure of all markers.

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

Proof. This is immediate from the definitions. ∎

LEMMA 3. *Let σ be a partially marked type. If σ is a derived type in $\Lambda_2^{\#}$*

$$\text{VAR} \qquad A \vdash x : \sigma \qquad (x : \sigma) \in A$$

$$\text{INST}^{\#} \qquad \frac{A \vdash M : \forall^{(\#)}\alpha.\sigma}{A \vdash M : \sigma[\alpha := (\tau)^{\#}]}$$

$$\text{GEN} \qquad \frac{A \vdash M : \sigma}{A \vdash M : \forall\alpha.\sigma} \qquad \alpha \notin \mathrm{FV}(A)$$

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

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

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

(i.e., there are a term M and an environment A such that $A \vdash_{\Lambda_2^{\#}} M : \sigma$), 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.

Proof. The property of partially marked types defined in the statement of the lemma is preserved by all the inference rules of $\Lambda_2^{\#}$ and holds for all types in the environment (by definition). ∎

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

DEFINITION 4 (( )$^{\bullet}$). We define a mapping that assigns to every partially marked type σ, an unmarked type $\sigma^{\bullet}$:

1. $\alpha^{\bullet} = \alpha$, 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 $\{\alpha_1, \alpha_2, ..., \alpha_n\}$, 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. For example,

$$(\forall\alpha\,\forall^{\#}\beta(\alpha\to\beta)\to\forall^{\#}\gamma\,\forall\delta(\gamma\to\delta))^{\bullet}=\forall\delta(\forall\alpha(\alpha\to\beta)\to\gamma\to\delta)$$

LEMMA 5. *For every $0\leqslant k\leqslant 2$ and for every partially marked type σ of rank k, $\sigma^{\bullet}\in S(k)$ and $FV(\sigma)\subseteq FV(\sigma^{\bullet})$.*

Proof. If σ is a partially marked type of rank k and σ' is an unmarked type obtained from σ by erasing all marked quantifiers, then obviously σ' is of rank k too, $FV(\sigma)\subseteq FV(\sigma')$, and $\sigma^{\bullet}=(\sigma')^{\bullet}$. Hence it suffices to prove the lemma for types σ without marked quantifiers, in which case we also prove that $FV(\sigma)=FV(\sigma^{\bullet})$. The proof is easy. Technically it proceeds by double induction: on k and on the structure of σ. For $\sigma\in R(0)$, the result is obvious, because $\sigma^{\bullet}=\sigma$.

Let $0\leqslant k<2$ and assume that the result holds for k and for all σ. For $\sigma\in R(k+1)$, we proceed by induction on the definition of types. If σ is a type variable, the result is obvious.

If $\sigma=\sigma_1\to\sigma_2\in R(k+1)$, then $\sigma_1\in R(k)$ and $\sigma_2\in R(k+1)$. We have $\sigma_1^{\bullet}\in S(k)$ by the induction hypothesis (on k), and $\sigma_2^{\bullet}=\forall\vec{\alpha}.\rho\in S(k+1)$ by the induction hypothesis (on the definition of types), where ρ is not a $\forall$-type. This means that $\sigma_1^{\bullet}\to\rho\in S(k+1)$ and $\forall\vec{\alpha}.(\sigma_1^{\bullet}\to\rho)=\sigma^{\bullet}\in S(k+1)$, as desired. Moreover,

$$\begin{aligned}FV(\sigma^{\bullet})&=FV(\sigma_1^{\bullet})\cup FV(\rho)-\{\vec{\alpha}\}\\&=FV(\sigma_1^{\bullet})\cup FV(\sigma_2^{\bullet})\\&=FV(\sigma_1)\cup FV(\sigma_2)=FV(\sigma).\end{aligned}$$

If $\sigma=\forall\alpha.\sigma_1\in R(k+1)$, then $\sigma_1\in R(k+1)$. We have $\sigma_1^{\bullet}\in S(k+1)$ by the induction hypothesis (on the definition of types), so that $\forall\alpha.\sigma_1^{\bullet}=\sigma^{\bullet}\in S(k+1)$, as desired. Moreover, $FV(\sigma^{\bullet})=FV(\sigma_1^{\bullet})-\{\alpha\}=FV(\sigma_1)-\{\alpha\}=FV(\sigma)$. ∎

LEMMA 6. *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. Consider the tree representation T_i of σ_i, $i=1,2$. All quantifiers appear along the rightmost path of T_i. We rename all bound variables, and permute all adjacent quantifiers, so that σ_1 and σ_2 are (syntactically) identical after erasure of all markers. The conclusion of the lemma follows from the definition of $(\ )^{\bullet}$ and Lemma 3. ∎

LEMMA 7. *Let σ be 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

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

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

LEMMA 8. *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 $A \vdash_{\Lambda_2^\#} M : \sigma$ then $A^\bullet \vdash_{\Lambda_2^-} M : \sigma^\bullet$.*

Proof. The proof is by induction on the length $n \geqslant 1$ of derivations $\mathscr{D}$ in $\Lambda_2^\#$:

$$\mathscr{D} = A_1 \vdash_{\Lambda_2^\#} M_1 : \sigma_1, \ldots, A_n \vdash_{\Lambda_2^\#} M_n : \sigma_n.$$

For $n = 1$, if $A_1 \vdash_{\Lambda_2^\#} M_1 : \sigma_1$ then clearly $A_1^\bullet \vdash_{\Lambda_2^-} M_1 : \sigma^\bullet$, by one application of VAR in both derivations.

For brevity, we write $\vdash$ instead of $\vdash_{\Lambda_2^\#}$ and $\vdash_{\Lambda_2^-}$ throughout this proof; it will be clear from the context whether the derivation is in $\Lambda_2^\#$ or in Λ_2^-.

If the nth step in $\mathscr{D}$, with $n \geqslant 2$, is an application of APP$^\#$, then we have in $\Lambda_2^\#$

$$A \vdash M : \sigma_1 \to \tau$$
$$A \vdash N : \sigma_2$$
$$A \vdash MN : \tau,$$

where $\sigma_1 \cong \sigma_2$. Let $\tau^\bullet = \forall\vec{\alpha}.\rho$, where ρ is not a $\forall$-type. Because σ_1, $\sigma_2 \in R(1)$, we have $\sigma_1^\bullet = \forall\vec{\beta}_1.\pi$ and $\sigma_2^\bullet = \forall\vec{\beta}_2.\pi$ by Lemma 6, where π is not a $\forall$-type, together with $\vec{\beta}_1 \subseteq \vec{\beta}_2$ or $\vec{\beta}_2 \subseteq \vec{\beta}_1$. Consider the second case, $\vec{\beta}_2 \subseteq \vec{\beta}_1$, the first case being treated similarly. All the variables in $\vec{\beta}_1 - \vec{\beta}_2$ are quantified in σ_1 and σ_2, and can therefore be taken disjoint from all free type variables in A (and $A^\bullet$ too), by α-conversion. By the induction hypothesis, we have in Λ_2^-

$$A^\bullet \vdash M : (\sigma_1 \to \tau)^\bullet = \forall\vec{\alpha}.((\forall\vec{\beta}_1.\pi) \to \rho) \tag{1}$$
$$A^\bullet \vdash N : \sigma_2^\bullet = \forall\vec{\beta}_2.\pi. \tag{2}$$

By k applications of INST$^-$ to the assertion (1), where $k = |\vec{\alpha}| \geqslant 0$, and l

applications of GEN to the assertion (2), where $l = |\vec{\beta}_1 - \vec{\beta}_2| \geqslant 0$, we obtain in Λ_2^-

$$A^\bullet \vdash M : (\forall \vec{\beta}_1 . \pi) \to \rho$$
$$A^\bullet \vdash N : \forall \vec{\beta}_1 . \pi.$$

By one application of APP, followed by k applications of GEN, we finally obtain in Λ_2^-

$$A^\bullet \vdash MN : \forall \vec{\alpha} . \rho = \tau^\bullet,$$

which is the desired conclusion.

If the nth step in $\mathscr{D}$, $n \geqslant 2$, is an application of ABS, then we have in $\Lambda_2^\#$

$$A[x : \sigma] \vdash M : \tau$$
$$A \vdash \lambda x . M : \sigma \to \tau.$$

Let $\tau^\bullet = \forall \vec{\alpha} . \rho$, where ρ is not a $\forall$-type. By the induction hypothesis, we have in Λ_2^- :

$$A^\bullet [x : \sigma^\bullet] \vdash M : \tau^\bullet = \forall \vec{\alpha} . \rho$$

By k applications of INST$^-$, where $k = |\vec{\alpha}| \geqslant 0$, followed by one application of ABS, followed by k applications of GEN, we obtain in Λ_2^-

$$A^\bullet \vdash \lambda x . M : \forall \vec{\alpha} . (\sigma^\bullet \to \rho) = (\sigma \to \tau)^\bullet,$$

which is the desired conclusion.

If the nth step in $\mathscr{D}$, $n \geqslant 2$, is an application of GEN, then we have in $\Lambda_2^\#$

$$A \vdash M : \sigma$$
$$A \vdash M : \forall \alpha . \sigma,$$

where $\alpha \notin FV(A)$. By the induction hypothesis, we have in Λ_2^-

$$A^\bullet \vdash M : \sigma^\bullet$$

and applying GEN once, with the fact that $FV(A) = FV(A^\bullet)$,

$$A^\bullet \vdash M : \forall \alpha . \sigma^\bullet = (\forall \alpha . \sigma)^\bullet,$$

as desired.

If the nth step in $\mathscr{D}$, $n \geqslant 2$, is an application of INST$^{\#}$, then we have in $\Lambda_2^{\#}$

$$\begin{aligned} &A \vdash M : \forall^{(\#)}\alpha.\sigma \\ &A \vdash M : \sigma[\alpha := \tau], \end{aligned} \tag{3}$$

where τ is totally marked. We can assume that $\alpha \notin FV(A)$. Consider the case when the outermost quantifier is marked, in the derived type of the assertion (3), the other case being treated similarly. By the induction hypothesis, we have in Λ_2^-

$$A^\bullet \vdash M : \sigma^\bullet.$$

Applying GEN once, we have in Λ_2^-

$$A^\bullet \vdash M : \forall\alpha.\sigma^\bullet,$$

and applying INST$^-$ once, with the fact that τ is a totally marked type and Lemma 7,

$$A^\bullet \vdash M : \sigma^\bullet[\alpha := \tau^\bullet] = (\sigma[\alpha := \tau])^\bullet$$

which is again the desired conclusion. ▮

THEOREM 9. *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 8. ▮

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:

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

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

$$(\dagger)\ \begin{cases} \text{no variable is bound more than once in } M, \\ \text{no variable occurs both bound and free in } M. \end{cases}$$

$$\text{VAR} \qquad A \vdash x : \sigma \qquad (x : \sigma) \in A$$

$$\text{INST} \qquad \frac{A \vdash M : \forall\alpha.\sigma}{A \vdash M : \sigma[\alpha := \tau]}$$

$$\text{GEN} \qquad \frac{A \vdash M : \sigma}{A \vdash M : \forall\alpha.\sigma} \qquad \alpha \notin \mathrm{FV}(A)$$

$$\text{APP} \qquad \frac{A \vdash M : \tau \to \tau', \quad A \vdash N : \tau}{A \vdash (M\ N) : \tau'}$$

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

$$\text{LET} \qquad \frac{A[x : \sigma] \vdash M : \tau \quad A \vdash N : \sigma}{A \vdash (\textbf{let } x = N \textbf{ in } M) : \tau}$$

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

(†) is satisfied by α-conversion. The system for typing **ML** terms is given in Fig. 4 (cf. Damas and Milner, 1982).

We 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[3] 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 two rules shown in Fig. 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 10. *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 the* INST *rule is applied only to variables; i.e., instead of* INST *the following more restrictive rule* INST^{var} *is used throughout the derivation:*

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

Proof. The proof is by induction on M. A substitution that corresponds to an application of INST to a term that is not a variable can be easily

[3] Identified as "the main limitation of the system" in R. Milner's original paper (1978, p. 356).

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

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

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

"pushed" all the way towards the leaves of the derivation tree. Eeasy details are left for the reader. ∎

There is a number of similar push-out lemmas in the literature (see Clément, Despeyroux, Despeyroux, and Kahn, 1986; Mitchell, 1988).

DEFINITION 11 (*act*). 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. $act(\lambda x.M) = x \cdot act(M)$
3. $act(MN) = \begin{cases} \varepsilon & \text{if } act(M) = \varepsilon \\ x_2 \cdots x_n & \text{if } act(M) = x_1 \cdots x_n, \text{ for some } n \geqslant 1 \end{cases}$
4. $act(\textbf{let } x = N \textbf{ in } M) = act(M)$.

The sequence $act(M)$ represents outstanding abstractions in M, i.e., those abstractions which are not "captured" by an application. It is the main technical tool to carry the proof of polynomial time reduction of Λ_2 typability to **ML** typability. Let us observe that due to our convention (†) at the beginning of this section, there are no repetitions of variables in $act(M)$.

DEFINITION 12 ($(\ )_x$). Let x be a variable and let M be an **ML** term. We define M_x, the effect of deleting λx in M. The inductive definition 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. $(\textbf{let } y = N \textbf{ in } M)_x = (\textbf{let } y = N_x \textbf{ in } M_x)$.

LEMMA 13. *Let M be an* **ML** *term such that $act(M) \neq \varepsilon$ and let x be the*

leftmost variable in $act(M)$. *Then for all types* τ, $\rho \in S(0)$ *and for every environment* A,

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

Proof. We prove the statement by induction on M. Let τ, $\rho \in S(0)$ be arbitrary open types. The above result holds vacuously when $M = x$ is a variable.

Let $M = \lambda x.N$. Then $M_x = N$ and $act(M) = x \cdot act(N)$. We have to show

$$A[x:\tau] \vdash_{\mathrm{ML}} N : \rho \qquad \text{iff} \qquad A \vdash_{\mathrm{ML}} \lambda x.N : \tau \to \rho \tag{4}$$

The implication from left to right in (4) follows from an application of the ABS rule. The opposite implication follows directly from Lemma 10.

Let $M = NP$ and let $act(N) = yx\cdots$. In that case we have to show

$$A[x:\tau] \vdash_{\mathrm{ML}} (N_x)(P_x) : \rho \qquad \text{iff} \qquad A \vdash_{\mathrm{ML}} NP : \tau \to \rho. \tag{5}$$

We prove the implication in (5) from left to right. The proof of the other implication, being similar, is omitted. Assume

$$A[x:\tau] \vdash_{\mathrm{ML}} (N_x)(P_x) : \rho.$$

By Lemma 10 we have

$$A[x:\tau] \vdash_{\mathrm{ML}} N_x : \sigma \to \rho \qquad \text{and} \qquad A[x:\tau] \vdash_{\mathrm{ML}} P_x : \sigma$$

for some $\sigma \in S(0)$. Since $act(N_x)$ ends with y, it follows, by the induction assumption, that

$$A[x:\tau][y:\sigma] \vdash_{\mathrm{ML}} N_{xy} : \rho.$$

Since $N_{xy} = N_{yx}$, applying the induction assumption to N_{yx} and to N_y we get

$$A[y:\sigma] \vdash_{\mathrm{ML}} N_y : \tau \to \rho$$

$$A \vdash_{\mathrm{ML}} N : \sigma \to \tau \to \rho.$$

Since x occurs among active variables of N it follows by our convention that x does not occur in P. Hence $P_x = P$ and we obtain

$$A \vdash_{\mathrm{ML}} NP : \tau \to \rho.$$

To complete the proof of Lemma 13 we consider the case when $M =$ (**let** $y = P$ **in** N) and $act(M) = act(N)$ starts with x. What we have to show now is

$$A[x:\tau] \vdash_{\mathrm{ML}} \textbf{let } y = P_x \textbf{ in } N_x : \rho \quad \text{iff} \quad A \vdash_{\mathrm{ML}} \textbf{let } y = P \textbf{ in } N : \tau \to \rho. \tag{6}$$

Assume the left side of (6). Again, since x occurs as an active variable in N it follows that x does not occur in P and $P_x = P$. By Lemma 10 we have

$$A[x:\tau][y:\sigma] \vdash_{\mathrm{ML}} N_x : \rho \qquad \text{and} \qquad A[x:\tau] \vdash_{\mathrm{ML}} P : \sigma \tag{7}$$

for some $\sigma \in S(1)$. Since x does not occur in P, and by the induction assumption we have from (7)

$$A[y:\sigma] \vdash_{\mathrm{ML}} N : \tau \to \rho \qquad \text{and} \qquad A \vdash_{\mathrm{ML}} P : \sigma, \tag{8}$$

the last condition immediately gives the right side in (6). The proof of the opposite implication is very similar and we omit it. ▮

DEFINITION 14 ($(\ )_L$). We define a mapping that assigns to every pure term M an **ML** term $(M)_L$:

1. $(x)_L = x$
2. $(\lambda x.M)_L = \lambda x.(M)_L$
3. $(MN)_L = \begin{cases} (M)_L\,(N)_L & \text{if } act(M) = \varepsilon \\ \textbf{let } x = (N)_L \textbf{ in } ((M)_L)_x & \text{if } act(M) \text{ starts with } x. \end{cases}$

Because $act(\textbf{let } x = N \textbf{ in } M) = act(M)$, by definition, it is easy to check that $act(M_L) = act(M)$ for all terms M.

LEMMA 15. *Let M be a pure term such that $act(M) = x_1 \cdots x_n$ for some $n \geq 0$. If $A \vdash_{A_2^-} M : \sigma$, for some environment A and $\sigma \in S(2)$, then*

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

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. This is a routine induction on the length of a derivation. We leave the details for the reader. ▮

LEMMA 16. *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 $\sigma = \sigma_1 \to \cdots \to \sigma_n \to \tau$, where $\sigma_1, \ldots, \sigma_n \in S(1)$ and $\tau \in S(0)$,*

$$A \vdash_{A_2^-} M : \sigma \qquad \textit{iff} \qquad A[x_1 : \sigma_1, \ldots, x_n : \sigma_n] \vdash_{\mathrm{ML}} ((M)_L)_{x_1 \cdots x_n} : \tau.$$

Proof. We prove the lemma by induction on pure terms M. If M is a variable x, then the conclusion of the lemma is

$$A \vdash_{A_2^-} x : \tau \qquad \text{iff} \qquad A \vdash_{\mathrm{ML}} x : \tau \tag{9}$$

which obviously holds since the derivation of the left side of (9) is in Λ_2^-, rather than in Λ_2.

Next, let us consider the case $M = \lambda y.N$, and let $act(M) = yx_1 \cdots x_n$. Take any $\rho, \sigma_1, ..., \sigma_n \in S(1)$ and $\tau \in S(0)$. Then

$$A \vdash_{\Lambda_2^-} \lambda y.N : \rho_1 \to \sigma_1 \to \cdots \to \sigma_n \to \tau \tag{10}$$

$$\text{iff} \quad A[y : \rho] \vdash_{\Lambda_2^-} N : \sigma_1 \to \cdots \to \sigma_n \to \tau \tag{11}$$

$$\text{iff} \quad A[y : \rho, x_1 : \sigma_1, ..., x_n : \sigma_n] \vdash_{\mathrm{ML}} ((N)_L)_{x_1 \cdots x_n} : \tau \tag{12}$$

Since in derivation in Λ_2^- the INST rule can be pushed to variables (by Lemma 10) it follows that (10) and (11) are equivalent. The equivalence of (11) and (12) follows from the induction assumption.

Now, let $M = NP$ and let $act(N) = \varepsilon$. Let $act(P) = y_1 \cdots y_m$, for some $m \geqslant 0$. Take any $\tau \in S(0)$ and assume

$$A \vdash_{\Lambda_2^-} NP : \tau. \tag{13}$$

It follows from Lemma 10 that there exists $\rho \in S(1)$ such that

$$A \vdash_{\Lambda_2^-} N : \rho \to \tau \qquad \text{and} \qquad A \vdash_{\Lambda_2^-} P : \rho \tag{14}$$

Since $\rho \in S(1)$, it follows from (14) and Lemma 15 that there are $\rho_0, ..., \rho_m \in S(0)$ such that $\rho = \forall \vec{\alpha}.(\rho_1 \to \cdots \to \rho_m \to \rho_0)$, and we may assume without loss of generality that none of the variables $\vec{\alpha}$ occurs free in A. Thus, by the induction assumption applied to (14) we get

$$A[y_1 : \rho_1, ..., y_m : \rho_m] \vdash_{\mathrm{ML}} ((P)_L)_{y_1 \cdots y_m} : \rho_0. \tag{15}$$

Since $act(P) = act((P)_L)$, and since the variables of $\vec{\alpha}$ do not occur free in A, by Lemma 13 we obtain from (15)

$$A \vdash_{\mathrm{ML}} (P)_L : \rho. \tag{16}$$

By (16), (14), and the induction hypothesis we conclude that

$$A \vdash_{\mathrm{ML}} (N)_L\,(P)_L : \tau.$$

Thus we have proven the left-to-right implication of

$$A \vdash_{\Lambda_2^-} NP : \tau \qquad \text{iff} \qquad A \vdash_{\mathrm{ML}} (N)_L\,(P)_L : \tau \tag{17}$$

for the case of $act(N) = \varepsilon$.

The proof of the right-to-left implication in (17) is very similar to the previous one.

As the last case in the proof we consider $M = NP$, where $act(N) =$

$x_1 \cdots x_{n+1}$, $(n \geqslant 0)$ and $act(P) = y_1 \cdots y_m$, $(m \geqslant 0)$. Let $\sigma = \sigma_1 \to \cdots \to \sigma_n \to \tau$, where $\sigma_1, ..., \sigma_n \in S(1)$, and $\tau \in S(0)$. Since $x_1, ..., x_{n+1}$ are active variables in N, it follows from the syntactic restriction (†) we stated at the beginning of this section that $x_1, ..., x_{n+1}$ do not occur (free or bound) in P and therefore in $(P)_L$. Thus $((P)_L)_{x_1 \cdots x_n} = (P)_L$. Since $E_{xy} = E_{yx}$ holds for every **ML** expression E satisfying the syntactic restriction (†), all we have to show is

$$A \vdash_{A_2^-} NP : \sigma$$
$$\text{iff} \quad A[x_1 : \sigma_1, ..., x_n : \sigma_n] \vdash_{\mathrm{ML}} \textbf{let } x_{n+1} = (P)_L \textbf{ in } ((N)_L)_{x_1 \cdots x_{n+1}} : \tau. \tag{18}$$

Let us assume

$$A \vdash_{A_2^-} NP : \sigma.$$

By Lemma 10 it follows that there is $\sigma_{n+1} \in S(1)$ such that

$$A \vdash_{A_2^-} N : \sigma_{n+1} \to \sigma \qquad \text{and} \qquad A \vdash_{A_2^-} P : \sigma_{n+1}. \tag{19}$$

Repeating essentially the same argument as in the proof of the previous case we conclude from (19), with the help of Lemma 13 and the induction assumption, that

$$A[x_1 : \sigma_1, ..., x_{n+1} : \sigma_{n+1}] \vdash_{\mathrm{ML}} ((N)_L)_{x_1 \cdots x_{n+1}} : \tau \text{ and } A \vdash_{\mathrm{ML}} (P)_L : \sigma_{n+1}.$$

Therefore

$$A[x_1 : \sigma_1, ..., x_n : \sigma_n] \vdash_{\mathrm{ML}} \textbf{let } x_{n+1} = (P)_L \textbf{ in } ((N)_L)_{x_1 \cdots x_{n+1}} : \tau.$$

The proof of the right-to-left implication in (18) is very similar. ▮

Given M, computing $act(M)$, $(M)_L$, and $(M)_x$ can be easily done in polynomial time. Hence, combining Lemma 15, Lemma 16, and Theorem 9, we obtain the following result:

THEOREM 17. *For every pure term M, if $act(M) = x_1 \cdots x_n$, for some $n \geqslant 0$, then*

$$M \textit{ is typable in } A_2 \qquad \textit{iff} \qquad ((M)_L)_{x_1 \cdots x_n} \textit{ is typable in ML.}$$

Thus, A_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 $M^{\sim}$.

DEFINITION 18 (( )$^\sim$). 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 19. *For every* **ML** *term* M, *environment* A, *and type* $\sigma \in S(1)$

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

Thus, by Theorem 9, **ML** *typability is polynomial-time reducible to* Λ_2 *typability.*

Proof. The proof of Theorem 19 is by induction on **ML** terms M. The left-to-right implication is a routine induction and we skip the details. For the right-to-left implication we prove inductively 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 $A \vdash_{\mathrm{ML}} M : \sigma$.

The proof of the claim is by induction on **ML** terms M. If M is a variable, then the conclusion obviously holds.

Let $M = \lambda x.N$ and assume that

$$A \vdash_{\Lambda_2^-} I(\lambda x.N^\sim) : \sigma \tag{20}$$

holds for some $\sigma \in S(2)$. By Lemma 10, without loss of generality we may assume that, except for a sequence of GEN rules, the last rule in the derivation of (20) is an APP rule; i.e., for some $\tau \in S(1)$,

$$A \vdash_{\Lambda_2^-} I : \tau \to \sigma' \tag{21}$$

and

$$A \vdash_{\Lambda_2^-} \lambda x.N^\sim : \tau, \tag{22}$$

where σ is obtained from σ' by generalization. It is not difficult to see that it follows from (21) and the restricted form of the instantiation rule in Λ_2^- that σ' must be an instance of τ via a substitution of some open types. Thus $\sigma' \in S(1)$.

By (22) we have that there exist types τ_1, τ_2 such that τ is obtained from $\tau_1 \to \tau_2$ by generalization and instantiation, and that

$$A[x : \tau_1] \vdash_{\Lambda_2^-} N^\sim : \tau_2. \tag{23}$$

Thus, by the induction assumption, we obtain

$$A[x:\tau_1] \vdash_{\mathrm{ML}} N:\tau_2. \tag{24}$$

Now, starting wtih (24) and following in **ML** the same sequence of instantiation and generalization rules as the one in Λ_2^- that produced (22) from (23), we get

$$A \vdash_{\mathrm{ML}} \lambda x.N:\tau.$$

Since σ' is an instance of τ, we conclude that

$$A \vdash_{\mathrm{ML}} \lambda x.N:\sigma',$$

hence

$$A \vdash_{\mathrm{ML}} \lambda x.N:\sigma.$$

Let $M = NP$ and

$$A \vdash_{\Lambda_2^-} N^\sim P^\sim : \sigma \tag{25}$$

for some $\sigma \in S(2)$. Again, by Lemma 10, without loss of generality we may assume that, except for a sequence of GEN rules, the last rule used in the derivation of (25) is an APP rule. Thus, for some $\tau \in S(1)$,

$$A \vdash_{\Lambda_2^-} N^\sim : \tau \to \sigma' \tag{26}$$

and

$$A \vdash_{\Lambda_2^-} P^\sim : \tau, \tag{27}$$

where σ is obtained from σ' by generalization. By (26) and the induction assumption, $\tau \to \sigma' \in S(1)$, hence $\sigma' \in S(1)$. By (26), (27) and the induction assumption we obtain

$$A \vdash_{\mathrm{ML}} N:\tau \to \sigma'$$

and

$$A \vdash_{\mathrm{ML}} P:\tau.$$

Thus,

$$A \vdash_{\mathrm{ML}} NP:\sigma'$$

and therefore

$$A \vdash_{\mathrm{ML}} NP:\sigma.$$

The case of $M = (\textbf{let } x = N \textbf{ in } P)$ is handled very similarly to the previous case. We leave the details for the reader. This completes the proof of the claim.

Since computing $M^{\sim}$ can be easily performed in polynomial time, Theorem 19 follows. ∎

THEOREM 20. *Given a pure term M and an environment A that assigns closed types in $S(1)$, it is decidable whether there exists an environment B and a type $\sigma \in R(2)$ such that $A \subseteq B$ and $B \vdash_{\Lambda_2} M : \sigma$.*

Proof. For $\mathscr{K}$ being one of the systems Λ_2, Λ_2^-, or **ML**, let $P_{\mathscr{K}}(A, M)$ be the property of an environment A and a term M that expresses the existence of an environment B and a type σ (of an appropriate rank) such that $A \subseteq B$ and $B \vdash_{\mathscr{K}} M : \sigma$.

If A is any environment that assigns only closed types in $S(1)$ then $A^* = A$. Let M be any pure term. Then, by Lemma 8

$$P_{\Lambda_2}(A, M) \quad \text{iff} \quad P_{\Lambda_2^-}(A, M)$$

By Lemma 15 and Lemma 16 we have

$$P_{\Lambda_2^-}(A, M) \quad \text{iff} \quad P_{\mathrm{ML}}(A, ((M)_L)_{x_1 \cdots x_n}),$$

where $act(M) = x_1 \cdots x_n$, $n \geqslant 0$. The decidability of P_{ML}, and therefore the conclusion of Theorem 20, follows from [5]. ∎

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 grammar

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

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:

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

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 assign types of rank k to all constants in the domain.

COROLLARY 21. *For every typing C of constants which assigns types in $S(1)$, typability in $\Lambda_2[C]$ is decidable.*

Proof. It follows that if C assigns types in $S(1)$ to constants, then for every term M with constants in the domain of C,

$$M \text{ is typable in } \Lambda_2[C] \quad \text{iff} \quad P_{\Lambda_2}(C', M') \text{ holds,} \tag{28}$$

where the predicate P_{Λ_2} is defined in the proof of Theorem 20, and C' and M' are an environment and a pure term, respectively, obtained from C and M by replacing the constant symbols by new variables. Since, by Theorem 20, P_{Λ_2} is decidable, hence typability in $\Lambda_2[C]$ is decidable too. ∎

Combining the proof of Theorem 20 with that of Corollary 21, we obtain the following generalization of Theorem 9.

COROLLARY 22. *For every typing C of constants which assigns types in $S(1)$, and for every pure term M with constants in the domain of C, M is typable in $\Lambda_2[C]$ iff M is typable in $\Lambda_2^-[C]$.*

Let us remind the reader that $\mathbf{ML}_1$ is an extension of **ML** defined in Fig. 5. Its main new feature is polymorphic abstraction.

COROLLARY 23. *If C is a typing of constants which assigns types in $S(1)$, then typability in $\mathbf{ML}_1[C]$, **ML** extended by polymorphic abstraction and constants in the domain of C, is decidable.*

Proof. Define a mapping that assigns to every **ML** term M with constants in C, the pure term $M^{\approx}$ with constants in C. (This mapping is even simpler than the one that precedes Theorem 19.)

1. $x^{\approx} = x$
2. $c^{\approx} = c$
3. $(\lambda x.M)^{\approx} = \lambda x.M^{\approx}$
4. $(MN)^{\approx} = M^{\approx}N^{\approx}$
5. **let** $x = N$ **in** $M = (\lambda x.M^{\approx})N^{\approx}$.

Now, clearly for every **ML** term M with constants in C, M is typable in $\mathbf{ML}_1[C]$ iff $M^{\approx}$ is typable in $\Lambda_2^-[C]$. Since, by Corollary 22, typability in $\Lambda_2[C]$ is equivalent to typability in $\Lambda_2^-[C^*]$ and $C = C^*$, the conclusion of Corollary 23 follows from Corollary 21. ∎

5. Typability in the Polymorphic λ-Calculus of Higher Ranks

In order to control the instantiation/generalization mechanism of the polymorphic type disciplines we introduce 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 $\sigma \leqslant_X \tau$, if there are a substitution S and a type τ' such that $S(\sigma) = \tau'$ (here S obviously acts only on the free variables of σ), $dom(S) \cap X = \varnothing$, and $\tau = \forall\alpha_1 \cdots \forall\alpha_n.\tau'$, where $\alpha_1, ..., \alpha_n \notin X$.

Let us define a sequence of open types $\tau_1, \tau_2, ...$ as follows. Let $\tau_1 = (\alpha \to \alpha)$ and let $\tau_{k+1} = (\tau_k \to \alpha)$. Let $k \geqslant 3$, 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 that for every $k \geqslant 3$, typability in $\Lambda_k[C_k]$ is undecidable. We prove it 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 Kfoury, Tiuryn, and Urzyczyn (to appear, b) (our $\mathbf{ML}^+$ here and in Kfoury, Tiuryn, and Urzyczyn (to appear, a) is denoted **ML**/1 in Kfoury, Tiuryn, and Urzyczyn (to appear, b)) and from the undecidability of the semi-unification problem (see Kfoury, Tiuryn, and Urzyczyn (to appear, c)) 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:

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

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

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

Definition 24 (( )$^\flat$). 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. It should be clear from this definition that computing $M^\flat$ from M can be done in polynomial time.

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 the 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 25. *For every set X of type variables,*

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

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

Proof. This follows immediately from the definitions. ∎

LEMMA 26. *Let M be a pure term with constants in C_k. If $A \vdash_{\Lambda_k[C_k]} cMy : \sigma$, then there exists $\tau \in R(0)$ such that $\tau \leqslant_{FV(A)} \sigma$, and $A \vdash_{\Lambda_k[C_k]} M : \tau$.*

Proof. Let $A \vdash_{\Lambda_k[C_k]} cMy : \sigma$, and let τ be the type assigned to M during this derivation when the APP rule is invoked for typing cM. Then the type of c must have been instantiated by τ, and therefore τ must be a quantifer-free type since otherwise the instantiated type of c would have had rank greater than k. Thus we have $A \vdash_{\Lambda_k[C_k]} M : \tau$ and $A \vdash_{\Lambda_k[C_k]} cM : \tau_k$. It follows that there exist types ρ_1, ρ_2 such that

$$\begin{gathered} \tau_k \leqslant_{FV(A)} \rho_1 \to \rho_2 \\ \rho_3 \leqslant_{FV(A)} \rho_1 \qquad \text{where} \quad (y : \rho_3) \in A, \\ A \vdash_{\Lambda_k[C_k]} cMy : \rho_2 \end{gathered} \tag{29}$$

and

$$\rho_2 \leqslant_{FV(A)} \sigma. \tag{30}$$

By (29) we have $\tau \leqslant_{FV(A)} \rho_2$. Hence, by (30) and Lemma 25(i) we conclude that $\tau \leqslant_{FV(A)} \sigma$. ∎

LEMMA 27. *If* $A \vdash_{A_k[C_k]} fMy : \sigma$, *then there is* $\tau \in R(1)$ *such that* $\tau \leqslant_{FV(A)} \sigma$ *and* $A \vdash_{A_k[C_k]} M : \tau \to \tau$.

Proof. Let $A \vdash_{A_k[C_k]} fMy : \sigma$ and let ρ be the type assigned to M in this derivation, when invoking the APP rule for typing fM. It follows from the definition of the type of f that for some type τ, $\rho = \tau \to \tau$ and $A \vdash_{A_k[C_k]} fM : \tau_{k-1}[\tau/\alpha]$. Since the latter type is in $R(k)$, it follows that $\tau \in R(1)$. ∎

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

LEMMA 28. *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* $A \vdash_{A_k^\#} M^\flat : \sigma$, *then* $A^\bullet \vdash_{\mathrm{ML}^+} M : \sigma^\bullet$.

Proof. The proof is by induction on M. Let M be a variable x, and assume that $A \vdash_{A_k^\#} cxy : \sigma$. Then, by Lemma 26, there is a type $\tau \in R(0)$ such that $A \vdash_{A_k^\#} x : \tau$ and

$$\tau \leqslant_{FV(A)} \sigma. \tag{31}$$

Thus, if $(x : \rho_1) \in A$, then $\rho_1 \leqslant_{FV(A)} \tau$. Since $\tau \in R(0)$, it follows that $\rho_1 \in S(1)$. Therefore $\rho_1^\bullet = \rho_1$ and

$$A^\flat \vdash_{\mathrm{ML}^+} x : \tau.$$

Since τ is an open type, it follows from (31) and Lemma 25(ii) that $\sigma^\flat$ is an $FV(A)$-instance of τ with open types. Hence $A^\flat \vdash_{\mathrm{ML}^+} x : \sigma^\bullet$.

Next we consider the case of M being $M_1 M_2$. Let $A \vdash_{A_k^\#} (cM_1^\flat y_1)(cM_2^\flat y_2) : \sigma$. Then there exist types τ and ρ such that

$$\begin{aligned} &\tau \leqslant_{FV(A)} \sigma \\ &A \vdash_{A_k^\#[C_k]} cM_1^\flat y_1 : \rho \to \tau \end{aligned} \tag{32}$$

and

$$A \vdash_{A_k^\#[C_k]} cM_2^\flat y_2 : \rho.$$

By Lemma 26 there are open types ρ_0, ρ_1, ρ_2 such that

$$\rho_1 \leqslant_{FV(A)} \rho \qquad \text{and} \qquad \rho_2 \to \rho_0 \leqslant_{FV(A)} \rho \to \tau \tag{33}$$

and

$$A \vdash_{A_k^\#[C_k]} M_1^\flat : \rho_2 \to \rho_0 \qquad \text{and} \qquad A \vdash_{A_k^\#[C_k]} M_2^\flat : \rho_1.$$

By the induction assumption we get

$$A^\bullet \vdash_{\mathrm{ML}^+} M_1 : \rho_2 \to \rho_0 \qquad \text{and} \qquad A^\bullet \vdash_{\mathrm{ML}^+} M_2 : \rho_1. \tag{34}$$

By Lemma 25(ii) and (33) we have

$$\rho_1 \leqslant_{FV(A)} \rho^\bullet \qquad \text{and} \qquad \rho_2 \to \rho_0 \leqslant_{FV(A)} \forall\vec{\alpha}(\rho^\bullet \to \tau'), \tag{35}$$

where $\tau^\bullet = \forall\vec{\alpha}\tau'$. By (34) and (35) we easily get

$$A^\bullet \vdash_{\mathrm{ML}^+} (M_1 M_2) : \tau^\bullet. \tag{36}$$

By (32) and Lemma 25(ii) we conclude that $\tau^\bullet \leqslant_{FV(A)} \sigma^\bullet$. Thus, by (36) we get

$$A^\bullet \vdash_{\mathrm{ML}^+} (M_1 M_2) : \sigma^\bullet.$$

Now we consider the case of M being $(\lambda x.N)$. Let $A \vdash_{A_k^\#[C_k]} c(\lambda x.N^\flat)\, y : \sigma$. By Lemma 26 there is an open type τ such that $\tau \leqslant_{FV(A)} \sigma$ and $A \vdash_{A_k^\#[C_k]} \lambda x.N^\flat : \tau$. Thus there are open types ρ_1, ρ_2 such that

$$A[x : \rho_1] \vdash_{A_k^\#[C_k]} N^\flat : \rho_2 \qquad \text{and} \qquad \rho_1 \to \rho_2 \leqslant_{FV(A)} \tau.$$

By the induction assumption we obtain

$$A^\bullet[x : \rho_1] \vdash_{\mathrm{ML}^+} N : \rho_2.$$

Hence

$$A^\bullet \vdash_{\mathrm{ML}^+} (\lambda x.N) : \rho_1 \to \rho_2.$$

Since $\rho_1 \to \rho_2 \leqslant_{FV(A)} \tau^\bullet \leqslant_{FV(A)} \sigma^\bullet$, we conclude that

$$A^\bullet \vdash_{\mathrm{ML}^+} (\lambda x.N) : \sigma^\bullet.$$

The last case in the induction step is $M = \mathbf{fix}\, x.N$. It is quite similar to the previous case. Let $A \vdash_{A_k^\#[C_k]} f(\lambda x.N^\flat)\, y : \sigma$. By Lemma 27 there is a type $\tau \in R(1)$ such that

$$\tau \leqslant_{FV(A)} \sigma \qquad \text{and} \qquad A \vdash_{A_k^\#[C_k]} (\lambda x.N^\flat) : \tau \to \tau. \tag{37}$$

Hence, there is $\rho \in R(1)$ such that

$$\rho \leqslant_{FV(A)} \tau \qquad \text{and} \qquad A[x:\rho] \vdash_{\Lambda_k^{\#}[C_k]} N^b : \rho. \tag{38}$$

By the induction assumption we get

$$A^*[x:\rho^*] \vdash_{\mathrm{ML}^+} N : \rho^*.$$

Hence

$$A^* \vdash_{\mathrm{ML}^+} (\mathbf{fix}\, x.N) : \rho^*.$$

By (37), (38), and Lemma 25(ii) we obtain the conclusion in the last inductive case of the proof. ∎

LEMMA 29. *For every environment A, $\mathbf{ML}^+$ term M, and a type σ, if $A \vdash_{\mathrm{ML}^+} M : \sigma$, then there is an environment $B \supseteq A$ such that $B \vdash_{\Lambda_k[C_k]} M^b : \sigma$.*

Proof. The proof is by an obvious induction on M and is therefore omitted. ∎

By Lemma 28 and Lemma 29 we can immediately establish an effective reduction of typability in $\mathbf{ML}^+$ to typability in $\Lambda_k[C_k]$.

THEOREM 30. *Let $k \geqslant 3$. For every term M of $\mathbf{ML}^+$, M is typable in $\mathbf{ML}^+$ iff M^b is typable in $\Lambda_k[C_k]$. Hence, for every $k \geqslant 3$, typability in $\Lambda_k[C_k]$ is undecidable.*

6. CONCLUDING REMARKS

We have considered an infinite hierarchy of type systems, each of them being a subsystem of the well-known Girard/Reynolds system F_2. These subsystems are obtained by imposing a constraint on the rank of all types used in the derivation. Λ_k is the system obtained from F_2 by restricting all derivations to contain type assertions whose rank is k. The union of all these subsystems gives precisely all of F_2.

We prove in this paper that the type-reconstruction problem for the system Λ_2, extended by an arbitrary typing of constants that assigns types in $S(1)$ (these are the universally polymorphic types), is decidable and is polynomial-time equivalent to the type-reconstruction problem for *ML* extended with a typing of constants. Λ_2 strictly extends *ML* with respect to typability power since it can type arbitrary abstractions in which the lambda-bound variable is typed by a universally polymorphic type. Thus in

Λ_2, the **let**-construct, which is the only source of polymorphism in *ML*, becomes syntactic sugar.

We have also shown that the situation dramatically changes as one moves from rank 2 to 3 and up. We prove to this effect that for every $k \geqslant 3$, there is a typing of constants that assigns types in $S(1)$, such that the type-reconstruction problem for Λ_k extended by this typing is undecidable. Unfortunately, this result does not give any clue as to whether type reconstruction for F_2, with or without constants, is decidable. Our decidability/undecidability proofs use in an essential way the apriori information about the rank of types.

There are two open problems which we are leaving unresolved in this paper. The first is whether the decidability result for Λ_2 with constants extends to any typing that assigns arbitrary types of rank 1, rather than universally polymorphic types as we assume in our paper. A typical example of a type that is of rank 1 but not in $S(1)$ is $\forall\alpha.(\alpha \to \forall\beta.(\beta \to \forall\gamma.(\alpha \to \beta \to \gamma)))$. The other open problem is whether type reconstruction is decidable for Λ_3, Λ_4, ... without any constants.

ACKNOWLEDGMENTS

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

RECEIVED September 24, 1990; FINAL MANUSCRIPT RECEIVED January 9, 1992

REFERENCES

BOEHM, H.-J. (1985), Partial polymorphic type inference is undecidable, *in* "Proceedings, 26th IEEE Symposium, Foundations of Computer Science," pp. 339–345.

CLÉMENT, D., DESPEYROUX, J., DESPEYROUX, T., AND KAHN, G. A simple applicative language: Mini-ML, *in* "Proceedings, ACM Conference on Lisp and Functional Programming," pp. 13–27.

COPPO, M., AND DEZANI-CIANCAGLINI, M. (1979), A new type assignment for λ-terms, *Arch. Math. Logik Grundlagenforschung* **19**, 139.

COPPO, M., DEZANI-CIANCAGLINI, M., AND VENERI, B. (1981), Functional characters of solvable terms, *Z. Math. Logik Grundlag. Math.* **27**, 45.

DAMAS, L., AND MILNER, R. (1982), Principle type schemes for functional programs, *in* "Proceedings, 9th ACM Symposium, Principles of Programming Languages," pp. 207–212.

FORTUNE, S., LEIVANT, D., AND O'DONNELL, M. (1983), The expressiveness of simple and second-order type structures, *Assoc. Comput. Mach.* **30**, 151.

GIANNINI, P. (1985), "Type-Checking and Type Deduction Techniques for Polymorphic Programming Languages," Technical Report, Department of Computer Science, CMU.

GIANNINI, P., AND RONCHI DELLA ROCCA, S. (1988), Characterization of typings in polymorphic type discipline, *in* "Proceedings of IEEE 3rd LICS," pp. 61–71.

GIANNINI, P., AND RONCHI DELLA ROCCA, S. (1991), Type inference in polymorphic type discipline, *in* "Proceedings of the International Conference on Theoretical Aspects of Computer Software, Tohoku University, Sendai, Japan" (T. Ito and A. R. Meyer, Eds.), pp. 18–37, Lecture Notes in Computer Science, Vol. 526, Springer-Verlag, Berlin/New York.

GIRARD, J.-Y. (1972), "Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur," Doctoral thesis, Université Paris VII.

HENGLEIN, F. (1988), Type inference and semi-unification, *in* "Proceedings, ACM Symposium, LISP and Functional Programming," pp. 184–197.

HENGLEIN, F., AND MAIRSON, H. G. (1991), The complexity of type inference for higher-order typed lambda calculi, *in* "Proceedings, 18th ACM Symposium on Principles of Programming Languages," pp. 119–130.

HINDLEY, J. R. (1969), The principal type-scheme of an object in combinatory logic, *Trans. Amer. Math. Soc.* **146**, 29.

KANELLAKIS, P. C., MAIRSON, H. G., AND MITCHELL, J. C. (to appear), Unification and ML type reconstruction, *in* "Computational Logic, Essays in Honor of Alan Robinson," MIT Press. Preliminary versions appeared under the respective titles Polymorphic unification and ML typing (by Kanellakis and Mitchell) *in* "Proceedings of POPL 1989," pp. 105–115, and Deciding ML typability is complete for deterministic exponential time (by Mairson), *in* "Proceedings of POPL 1990," pp. 382–401.

KFOURY, A. J., TIURYN, J., AND URZYCZYN, P. (1988), A proper extension of ML with an effective type-assignment, *in* "Proceedings, 15th ACM Symposium, Principles of Programming Languages," pp. 58–69.

KFOURY, A. J., TIURYN, J., AND URZYCZYN, P. (to appear, a), Type-reconstruction in the presence of polymorphic recursion, *in* "ACM Transactions on Programming Languages and Systems." Part of the results of this paper were presented in Computational consequence and partial solutions of a generalized unification problem, *in* "Proceedings of 4th IEEE LICS, 1989," pp. 98–105.

KFOURY, A. J, TIURYN, J., AND URZYCZYN, P. (to appear, b), The undecidability of the semi-unification problem, *Inform. Comput.* A preliminary version appeared *in* "Proceedings of 22th ACM STOC," pp. 468–477, May 1990.

KFOURY, A. J., TIURYN, J., AND URZYCZYN, P. (to appear, c), An analysis of ML typability, *J. Assoc. Comput. Mach.* A preliminary version appeared *in* "Proceedings, 15th Colloquim on Trees in Algebra and Programming, CAAP '90" (Arnold, Ed.), Lecture Notes in Computer Science, Vol. 431, Springer-Verlag, Berlin/New York, 1990.

LEISS, H. (1987), On type inference for object-oriented programming languages, *in* "Proceedings, 1st Workshop on Computer Science Logic" (Börger, Büning, and Richter, Eds.), Lecture Notes in Computer Science, Vol. 329, Springer-Verlag, Berlin/New York.

LEIVANT, D. (1983), Polymorphic type inference, *in* "Proceedings of 10th ACM Symposium, Principles of Programming Languages," pp. 88–98.

LEIVANT, D. (1989), Stratified polymorphism (extended summary), *in* "Proceedings, 4th IEEE Symposium Logic in Computer Science," pp. 39–47.

MCCRACKEN, N. (1984), The typechecking of programs with implicit type structure, *in* "Semantics of Data Types" (Kahn, McQueen, and Plotkin, Eds.), pp. 301–315, Lecture Notes in Computer Science, Vol. 173, Springer-Verlag, Berlin/New York.

MILNER, R. (1978), A theory of type polymorphism in programming, *J. Comput. System Sci.* **17**, 348.

MITCHELL, J. C. (1988), Polymorphic type inference and containment, *Inform. Comput.* **76**, 211.

MYCROFT, A. (1984), Polymorphic type schemes and recursive definition, *in* "International Symposium on Programming" (Paul and Robinet, Eds.), Lecture Notes in Computer Science, Vol. 167, Springer-Verlag, Berlin/New York.

PFENNING, F. (1988), Partial polymorphic type inference and higher-order unification, *in* "Proceedings of Lisp and Functional Programing Conference." pp. 153–163.

REYNOLDS, J. (1974), Towards a theory of type structure, *in* "Proceedings, Colloque sur la Programmation," pp. 408–425, Lecture Notes in Computer Science, Vol. 19, Springer-Verlag, Berlin/New York.

TIURYN, J. (1988), Representability of arithmetic functions in fragments of second-order λ-calculus, manuscript.

TIURYN, J. (1990), Type inference problems: A survey, *in* "Proceedings, International Symposium on Mathematical Foundations of Computer Science, Banska Bystrica, Czechoslovakia" (B. Rovan, Ed.), pp. 105–120, Lecture Notes in Computer Science, Vol. 452, Springer-Verlag, Berlin/New York.

WAND, M. (1987), Complete type inference for simple objects, *in* "Proceedings, 2nd IEEE Symposium on Logic in Computer Science," pp. 37–44. See alo Corrigendum: Complete type inference for simple types, *in* "Proceedings, 3rd IEEE Symposium on Logic in Computer Science," p. 132.