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

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

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

December 1, 1993

Boston University Computer Science Department
Technical Report 93-017

Abstract

We study the problem of type inference for a family of polymorphic type disciplines containing the power of Core-**ML**. This family comprises all levels of the stratification of the second-order lambda-calculus by "rank" of types. We show that typability is an undecidable problem at every rank $k \geq 3$ of this stratification. While it was already known that typability is decidable at rank ≤ 2, no direct and easy-to-implement algorithm was available. To design such an algorithm, we develop a new notion of reduction and show how to use it to reduce the problem of typability at rank 2 to the problem of acyclic semi-unification. A by-product of our analysis is the publication of a simple solution procedure for acyclic semi-unification.

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

1 Introduction

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

The usefulness of a particular polymorphic type system depends very much on how feasible the tasks of type checking and type inference are. We define these concepts in terms of the untyped λ-calculus, which we use as our pure functional programming language throughout this paper. By *type checking* we mean the problem of deciding, given a λ-term M and a type τ, whether τ is one of the types that may be derived for M by the type system under consideration. By *type inference* we mean the problem of finding a type derivable for a λ-term in the type system. The problem of type inference involves several issues:

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

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

(2) How efficiently can typability be decided? How efficiently can type inference be performed?

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

The answers to these questions determine how feasible the type system is to implement.

In addition to the feasibility of a particular polymorphic type system, its usefulness also depends on how much flexibility the type system gives the programmer. Although the polymorphism of **ML** is useful, it is too weak to assign types to some program phrases that are natural for programmers to write. To overcome these limitations researchers have investigated the feasibility of type systems whose typing power is a superset of that of **ML**. Over the years, this line of research has dealt with various polymorphic type systems for functional languages and λ-calculi, in particular the powerful type system of the Girard/Reynolds second-order λ-calculus [Gir72, Rey74], which we will call by its other name, System **F**. In the long quest to settle the type checking and typability problems for **F**, researchers have also considered the problem for **F** modified by various restrictions. Multiple stratifications of **F** have been proposed, e.g. by depth of bound type variable from binding quantifier in [GRDR91] and by limiting the number of generations of instantiation of quantifiers themselves introduced by instantiation in [Lei91]. One natural restriction which we consider in this paper results from stratifying **F** according to the "rank" of types allowed in the typing of λ-terms and restricting the rank to various finite values (introduced in [Lei83] and further studied in [McC84, KT92]). All of these systems improve on the expressive power of **ML**.

Unfortunately, it is often the case that the more flexible and powerful a particular polymorphic type system is, the more likely that it will be infeasible to implement. As discouraging examples,

the problems of typability and type checking for many of the polymorphic type systems mentioned above have recently been proven undecidable. Type checking and typability were shown to be undecidable for System **F** (cf. recent results submitted for publication elsewhere in [Wel93]) and for its very powerful extension, System $\mathbf{F}_\omega$ [Urz93]. Other related systems that are not exactly extensions of **ML** have also recently been proven to have undecidable typability, i.e. System $\mathbf{F}_\leq$ which relates to object-oriented languages [Pie92], and the $\lambda\Pi$-calculus which relates to extensions of λ-Prolog [Dow93].

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

Contributions of This Paper. We can now firmly establish the boundary for decidability of typability and type checking within the stratification of System **F** by rank of types. The two problems are undecidable for every fragment of **F** of rank ≥ 3 and are decidable for rank ≤ 2. The undecidability of type checking at rank ≥ 3 can be seen by observing that the proof for the undecidability of type checking in **F** in [Wel93] requires only rank-3 types.[1] The undecidability of typability at rank ≥ 3 results from the fact that the constants c and f defined in section 5 of [KT92] can be encoded using the methods of [Wel93] in System Λ_3 (the rank-3 fragment of **F**) and from Theorem 30 of [KT92]. We give this encoding in this paper. Since it was already known from [KT92] that typability is decidable for System Λ_2 (the rank-2 fragment of **F**), we know exactly where the boundary of decidability for typability lies. These circumstances lead us to look for a simple and direct algorithm for type inference within Λ_2.

The existing proof that typability is decidable for System Λ_2 uses a succession of several reductions to the typability problem in **ML** and results in a type inference algorithm that is neither simple nor easy to understand. In this paper, we give a simpler and more direct algorithm for the decidable case of typability in Λ_2. We first prove that Λ_2 is equivalent to a restriction named System $\Lambda_2^{-\prime *}$ having many convenient properties. We then develop a notion of reduction named θ which converts λ-terms into a form which is more amenable to type inference but which also preserves every λ-term's set of derivable types in $\Lambda_2^{-\prime *}$. The type inference problem in $\Lambda_2^{-\prime *}$ for a λ-term in θ-normal form is easily converted into an acyclic semi-unification problem. Finally, we give a simple algorithm for solving acyclic semi-unification problem. The complexity of the whole procedure is the same as that of type inference in **ML**.

We omit all proofs of all lemmas and theorems in this conference report to remain within the page limit. A later extended version of this paper will clarify the relationship between **ML**-typability and typability in Λ_2 and discuss the issues of type checking and principal types in Λ_2.

Acknowledgements. A number of definitions used in this paper were lifted from [KT92, KTU90, KTU93].

2 System Λ_k and System Λ_2^-

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

[1] In second version of report to appear mid-December 1993.

pure terms of the λ-calculus are assigned types, rather than the "Church view" where terms and types are defined simultaneously to produce typed terms.

The set of all λ-terms Λ is built from the set of λ-term variables $\mathcal{V}$ using application and abstraction as specified by the usual grammar $\Lambda ::= \mathcal{V} \mid (\Lambda\,\Lambda) \mid (\lambda\mathcal{V}.\Lambda)$. We use small Roman letters towards the end of the alphabet as metavariables ranging over $\mathcal{V}$ and capital Roman letters as metavariables ranging over Λ. When writing λ-terms, application associates to the left so that $MNP \equiv (MN)P$. The scope of "$\lambda x.$" extends as far to the right as possible, while the scope of "λx" without the "." covers as little as possible.

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

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

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

We define a notation for specifying many quantifiers concisely. For type σ and set of type variables $X \subseteq \mathrm{FTV}(\sigma)$, the shorthand notation $\forall X.\sigma$ is defined so that $\forall\varnothing.\sigma = \sigma$ and $\forall(X \cup \{\alpha\}).\sigma = \forall\alpha.\forall(X - \{\alpha\}).\sigma$. This defines just one type because we assume the order of quantifiers does not distinguish two types. We may use $\vec{\alpha}$ to stand for a sequence of type variables α_1, ..., α_n. We allow $\vec{\alpha}$ to be treated as a set or as a comma-separated sequence as is most convenient, so $\forall\vec{\alpha}.\sigma$ has the expected meaning. The notation $\forall.\sigma$ means $\forall(\mathrm{FTV}(\sigma)).\sigma$.

To define System Λ_k, we will use the following inductive stratification of types. First define $R(0)$ as the set of open types, i.e. types not mentioning $\forall$. Then, for all $k \geq 0$, define $R(k+1)$ by the grammar $R(k+1) ::= R(k) \mid (\,R(k){\to}R(k+1)\,) \mid (\,\forall\,\mathbb{V}.\,R(k+1)\,)$. We say that $R(k)$ is the set of types of *rank* k. For example, $\forall\alpha.\alpha{\to}\forall\beta.\alpha{\to}\beta$ is a type within rank 1 and $(\forall\alpha.\alpha{\to}\alpha){\to}\forall\beta.\beta$ is a type within rank 2 but not within rank 1. Our definition of rank is equivalent to the notion of

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

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

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

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

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

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

rank introduced in [Lei83]. Since $R(k) \subseteq R(k+1)$ it follows that if a type σ is within rank k, then it is within every rank $n \geq k$. Observe that performing the substitution $\sigma[\alpha{:=}\tau]$ may not preserve rank. The resulting rank depends on the rank of τ and how deep in the negative scope of $\to$ the free occurrences of α in σ are.

To define Λ_2^-, we will use subsets of the type sets $R(0)$, $R(1)$, and $R(2)$ called $S(0)$, $S(1)$, and $S(2)$. Let $S(0) = R(0)$ be the set of all open types. Let $S(1)$ be the set of all types of the form $\forall\vec{\alpha}.\sigma$, where $\sigma \in S(0)$. Let $S(2)$ be the set of all types of the form $\forall\vec{\alpha}.\sigma_1 \to \cdots \to \sigma_m \to \tau$, where $\sigma_1, \ldots, \sigma_m \in S(1)$ and $\tau \in S(0)$.

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

We define System **F** to be the type system that can derive types for λ-terms using the inference rules presented in Figure 1 with no other restrictions. For every $k \geq 0$, we define Λ_k as the restriction of **F** which allows only assertions within rank $\leq k$ to be derived. We define System Λ_2^- as a restriction of System Λ_2 where the two differences are that (i) in Λ_2^- all assertions must

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

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

have types in $S(2)$ (thus all assigned types are in $S(1)$ and all derived types in $S(2)$) and (ii) that the inference rule INST of Λ_2 is replaced by the rule INST$^-$ described in Figure 2. Theorem 9 in [KT92] shows that Λ_2^- types the same set of terms as Λ_2 with very similar types. Since Λ_2^- is as powerful as Λ_2 and since its restrictions make analysis of type inference easier, we will use it instead of Λ_2 in this paper.

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

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

Section 5 of [KT92] introduces System $\Lambda_k[C_k]$ for each $k \geq 3$ and Theorem 30 of the same paper proves that typability is undecidable for $\Lambda_k[C_k]$ for $k \geq 3$. The original definition of $\Lambda_k[C_k]$ defined it based on Λ_k by adding two constants, c and f, with predefined types $\phi_{c,k}$ and $\phi_{f,k}$. A simple alternate definition is to declare that $A \vdash M : \tau$ is derivable in $\Lambda_k[C_k]$ if and only if $A \cup \{c{:}\phi_{c,k}, f{:}\phi_{f,k}\} \vdash M : \tau$ is derivable in Λ_k.

The analysis for $\Lambda_3[C_3]$ goes as follows. For $\Lambda_3[C_3]$ the types of the constants c and f are $\phi_{c,3} = \forall\alpha.\alpha{\to}(((\alpha{\to}\alpha){\to}\alpha){\to}\alpha)$ and $\phi_{f,3} = \forall\alpha.(\alpha{\to}\alpha){\to}((\alpha{\to}\alpha){\to}\alpha)$. We construct a context $C_3[\cdot]$ with one hole:

$$
\begin{aligned}
J_i[\cdot] &\equiv (\lambda y_i.(\lambda z_i.r(y_i y_i(y_i z_i))))(\lambda x_i.\mathbf{K}x_i(\mathbf{K}(x_i(x_i r))[\cdot]))(\lambda w_i.w_i w_i)\\
D[\cdot] &\equiv (\lambda f.r(x_1(f x_1 x_1))(x_2(f x_2 x_2))[\cdot])(\lambda u.\lambda v.u(v(u(ur))))\\
E[\cdot] &\equiv (\lambda t.r(x_1(t x_1(x_1 r)(f x_1)))(x_2(t x_2(x_2 r)(f x_2)))[\cdot])(\lambda p.\lambda q.\lambda s.\mathbf{K}(p(pq))(p(sp)))\\
G[\cdot] &\equiv (\lambda c.r(x_1(c(x_1 r)(f x_1)))(x_2(c(x_2 r)(f x_2)))[\cdot])(tr)\\
C_3[\cdot] &\equiv \lambda r.J_1[J_2[D[E[G[\cdot]]]]]
\end{aligned}
$$

Using the methods of [Wel93], it can be seen that this context can be typed in Λ_3 and in any typing of this context (with any λ-term placed in the hole), the variables c and f must be assigned the types $\phi_{c,3}$ and $\phi_{f,3}$.

Since for each $k > 3$ a context $C_k[\cdot]$ having the same properties with respect to Λ_k can be constructed, we have this result:

Theorem 3.1 *For any type assignment A, there exists a type τ such that $A \cup \{c{:}\phi_{c,k}, f{:}\phi_{f,k}\} \vdash_{\Lambda_3} M : \tau$ is derivable if and only if there exists a type τ' such that $A \vdash_{\Lambda_3} C_k[M] : \tau'$ is derivable. Thus, the problem of typability for $\Lambda_k[C_k]$ for $k \geq 3$ is reducible to the problem for Λ_k. Therefore, typability is undecidable for Λ_k for every $k \geq 3$.*

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

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

Definition 4.1 (*act*) (Taken from [KT92].) Let us define, by induction on λ-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. $act(MN) = \begin{cases} \varepsilon & \text{if } act(M) = \varepsilon \\ x_2 \cdots x_n & \text{if } act(M) = x_1 \cdots x_n, \text{ for some } n \geq 1 \end{cases}$

Let us observe that due to our conventions, there are no repetitions of variables in $act(M)$. The sequence $act(M)$ represents outstanding abstractions in M, i.e. those abstractions which have not been "captured" by an application. For each application subterm $Q \equiv RS$ in a λ-term M where $act(R) = x \cdots$, there is an abstraction subterm $N \equiv (\lambda x.P)$ within R (possibly R itself). In this case, we say that the subterms N, Q, and S are *companions*, specifically, N is the *companion abstraction*, Q the *companion application*, and S the *companion argument* of the others. In this case, if N is ever β-reduced, its argument will be S or S's β-descendent. If $N \equiv R$, i.e. $Q \equiv NS$, then we say that they are *adjacent companions* and it is the case that they are a β-redex. A set of non-adjacent companions represents a "potential" β-redex in a λ-term whose presence can be detected by simple inspection without β-reduction. Companions turn out to have convenient properties in Λ_2^-.

Definition 4.2 $(( )^\lambda)$ For a λ-term M, we define $(M)^\lambda$ as the effect of traversing M and labeling each of its abstraction subterms with an index $i \in \{1,2,3\}$, depending on the subterm's position and whether it has companions. $(M)^\lambda$ is defined in terms of an auxiliary function *label* which takes as parameters a λ-term, a set of variables, and an index. The inductive definition of *label* follows for $i \in \{1,2,3\}$:

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

We then finish the definition by saying that $(M)^\lambda = label(M, act(M), 2)$.

Informally, labeling the λ-term M affects each abstraction subterm N as follows. If N has companions, then it is labelled with λ^1. If N does not have companions, then it is labelled with λ^2 if there is no subterm $P = LR$ of M such that N lies within R, the right subterm. Otherwise N is labelled with λ^3. When dealing with a labelled λ-term M after this point, we will assume that the labeling is the result of the $( )^\lambda$ operator and not any arbitrary labeling, i.e. we assume that either $M = (N)^\lambda$ or $M \subset (N)^\lambda$ for some unlabelled λ-term N.

Lemma 4.3 *If $\mathcal{D}$ is a derivation in Λ_2^- that types the λ-term M, and there is an abstraction subterm $(\lambda x.N)$ in M, and there is a subterm (PQ) in M such that x appears in $act(Q)$, and there is an assertion $A \cup \{x{:}\sigma\} \vdash N : \tau$ in $\mathcal{D}$, then $\sigma \in S(0)$. Restated more informally, the bound variable of a companionless, λ^3-labelled abstraction must be assigned a monomorphic type.*

Lemma 4.4 *If in Λ_2^- there is a derivation $\mathcal{D}$ ending with the assertion $A \vdash M : \tau$, then for any type variable substitution $[\vec{\alpha}{:=}\vec{\chi}]$, it is the case that there is a derivation $\mathcal{D}'$ ending with the assertion $A[\vec{\alpha}{:=}\vec{\chi}] \vdash M : \tau[\vec{\alpha}{:=}\vec{\chi}]$ and, furthermore, $\mathcal{D}$ and $\mathcal{D}'$ are of the same length and there is a one-to-one correspondence between rule applications in both derivations.*

Lemma 4.4 is used by Lemma 4.5. For Lemma 4.5, let us temporarily suppose that quantifiers introduced into types by the GEN rule are marked. For example, from the assertion $A \vdash M : \tau$ where $\alpha \notin \mathrm{FTV}(A)$ we can derive using GEN the assertion $A \vdash M : \forall^\flat \alpha.\tau$. These markers on quantifiers do not affect the behavior of the inference rules; they merely allow us to precisely phrase the lemma.

Lemma 4.5 *If in Λ_2^- there is a derivation $\mathcal{D}$ ending with the assertion $A \vdash M : \tau$, then there is a derivation $\mathcal{D}'$ ending with the same assertion such that there is no use of the* INST *rule whose premise is an assertion of the form $B \vdash N : \forall^\flat \alpha.\rho$. In plainer English, we can assume that quantifiers introduced by* GEN *are never instantiated.*

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

Figure 3: Inference Rules of System $\Lambda_2^{-,*}$.

Lemma 4.6 *If $\mathcal{D}$ is a derivation in Λ_2^- that types the λ-term M, and $\mathcal{D}$ includes the assertion $A \vdash N : \forall\alpha.\tau$, and there are no subsequent assertions in $\mathcal{D}$ for the subterm N that are derived from this assertion, then either $N = M$ or there is a subterm (PN) in M where $act(P) \neq \varepsilon$. Rephrased, the only proper subterms for which the final derived type may be a $\forall$-type are companion arguments.*

Lemma 4.7 results from Lemmas 4.5 and 4.6.

Lemma 4.7 *If $\mathcal{D}$ is a derivation in Λ_2^- that types the λ-term M, and $\mathcal{D}$ includes the assertion $A \vdash N : \forall\alpha.\tau$ as a consequence of the* GEN *rule, then N is a companion argument.*

Lemma 4.8 *If in Λ_2^- there is a derivation $\mathcal{D}$ ending with the assertion $A \vdash M : \tau$, then there is a derivation $\mathcal{D}'$ ending with the same assertion such that if the assertion $B \vdash N : \sigma$ in $\mathcal{D}'$ is the consequence of the* INST *rule, then $N \in \mathcal{V}$, i.e. N is a variable. In other words, we can assume all uses of the* INST *rule occur at the leaves of the derivation (viewing the derivation as a tree).*

We now define the new System $\Lambda_2^{-,*}$ to formally include the restrictions proven by the previous lemmas into a type system. We present the inference rules for $\Lambda_2^{-,*}$ in Figure 3. As in Λ_2^-, all assertions are required to be within rank 2.

Theorem 4.9 *$A \vdash_{(\Lambda_2^-)} M : \tau$ holds if and only if $A \vdash_{(\Lambda_2^{-,*})} (M)^\lambda : \tau$ holds, i.e. every Λ_2^- typing is equivalent to a $\Lambda_2^{-,*}$ typing and vice versa.*

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

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

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

- θ_1 transforms a subterm of the form $(((\lambda^1 x.N)P)Q)$ to $((\lambda^1 x.NQ)P)$.

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

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

We use the notation θ_i where $i \in \{1,2,3,4\}$ to stand for one of θ_1, θ_2, θ_3, or θ_4. We define $\theta_{i,j}$ to be $\theta_i \cup \theta_j$ and define $\theta = \theta_{1,2,3,4}$. Since these transformations are all notions of reduction, $\rightarrow_{\theta_1}$, $\rightarrow_{\theta_{1,2}}$, $\rightarrow_\theta$, etc., have the expected meaning.

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

We now prove a variety of useful properties of θ-reduction.

Lemma 5.2 *Let M be in θ-normal form. M is of the form:*

$$\lambda^2 x_1.\lambda^2 x_2.\ldots.\lambda^2 x_m.(\lambda^1 y_1.(\lambda^1 y_2.(\ldots((\lambda^1 y_n.T_{n+1})T_n)\ldots))T_2)T_1$$

where $m \geq 0$, $n \geq 0$, and where T_1, ..., T_{n+1} are λ-terms in β-normal form. Furthermore, any abtractions within T_i for $1 \leq i \leq n+1$ are λ^3-labelled. Thus, all λ^1-labelled abstractions belong to β-redexes, i.e. there are no non-adjacent companions.

The λ-term M detailed in Lemma 5.2 can also be viewed as the following **ML** term:

$$\textbf{fn } x_1 \Rightarrow \textbf{fn } x_2 \Rightarrow \ldots \Rightarrow \textbf{fn } x_m \Rightarrow \textbf{let } y_1 = T_1 \textbf{ in let } y_2 = T_2 \textbf{ in } \ldots \textbf{let } y_n = T_n \textbf{ in } S$$

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

Lemma 5.4 *θ-reduction always terminates, i.e. it is strongly normalizing.*

Lemma 5.5 *We can assume that the type assigned to the bound variable of a λ^1-abstraction which is the function of a β-redex has no free type variables that are not also free somewhere else in the type assignment.*

Lemma 5.5 is used by Lemma 5.6.

Lemma 5.6 *If θ_1, θ_2, θ_3, or θ_4 transform M into N in one step, then with any particular type assignment, both M and N are typable with the same types in $\Lambda_2^{-,*}$. In other words, if $M \rightarrow_\theta N$, then in $\Lambda_2^{-,*}$ it holds that $A \vdash M : \pi$ is derivable if and only if $A \vdash N : \pi$ is derivable. As a result, $A \vdash_{(\Lambda_2^{-,*})} M : \tau$ is true if and only if $A \vdash_{(\Lambda_2^{-,*})} \theta\text{-}nf(M) : \tau$ is true.*

Lemma 5.7 *$act(\theta\text{-}nf((M)^\lambda)) = act(M)$.*

VAR^{θ}	$A \vdash x : \tau$	$A(x) \preceq^{0} \tau, \quad \tau \in S(0)$
APP^{θ}	$\dfrac{A \vdash M : \sigma \to \tau, \qquad A \vdash N : \sigma}{A \vdash (M\ N) : \tau}$	$\sigma, \tau \in S(0), \quad act(M) = \varepsilon$
LET^{θ}	$\dfrac{A \cup \{x{:}\forall.\sigma\} \vdash M : \tau, \qquad A \vdash N : \sigma}{A \vdash ((\lambda^{1} x.M)\ N) : \tau}$	$\sigma, \tau \in S(0)$
$\mathrm{ABS}^{\theta,1}$	$\dfrac{A \cup \{x{:}\forall.\sigma\} \vdash M : \tau}{A \vdash (\lambda^{1} x.M) : (\forall.\sigma) \to \tau}$	$\tau \in S(0)$
$\mathrm{ABS}^{\theta,2}$	$\dfrac{A \cup \{x{:}\bot\} \vdash M : \tau}{A \vdash (\lambda^{2} x.M) : \bot \to \tau}$	
$\mathrm{ABS}^{\theta,3}$	$\dfrac{A \cup \{x{:}\sigma\} \vdash M : \tau}{A \vdash (\lambda^{3} x.M) : \sigma \to \tau}$	$\sigma, \tau \in S(0)$

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

Lemma 5.8 (*From* [KT92].) *In* $\Lambda_2^{-,*}$, *if* $A \vdash M : \rho$ *is derivable and* $|act(M)| = n$, *then* $\rho = \forall\vec{\alpha}.\sigma_1 \to \ldots \to \sigma_n \to \tau$ *and* $\vec{\sigma} \in S(1)$ *and* $\tau \in S(0)$.

Lemma 5.9 *We can always assign the type* $\bot = \forall\alpha.\alpha$ *to the bound variable of a companionless,* λ^2*-labelled abstraction without affecting the whole* λ*-term's typability.*

Lemma 5.10 *Under the restriction that the outermost type assignment assigns the type* $\bot$ *to all variables, we can always assign universally closed types to the bound variables of every* λ^1*-labelled abstraction without affecting the whole* λ*-term's typability.*

We now define System $\Lambda_2^{-,*,\theta}$ to take advantage of the typing properties of λ-terms in θ-normal form in $\Lambda_2^{-,*}$. System $\Lambda_2^{-,*,\theta}$ is intended to be used only for θ-normal forms; its behavior on other λ-terms has not been investigated. The inference rules for $\Lambda_2^{-,*,\theta}$are presented in Figure 4. As with $\Lambda_2^{-,*}$, assigned types must be in $S(1)$ and derived types must be in $S(2)$.

Theorem 5.11 *Typability and type inference in* $\Lambda_2^{-,*}$ *are reducible to the same problems in* $\Lambda_2^{-,*,\theta}$. *For a labelled* λ*-term* M *where* $|act(M)| = n$, *if* $A \vdash_{(\Lambda_2^{-,*})} M : \forall\vec{\alpha}.\sigma_1 \to \cdots \to \sigma_n \to \tau$ *holds, then using the type assignment* B *that maps all variables to type* $\bot$ *it is the case that* $B \vdash_{(\Lambda_2^{-,*,\theta})} \theta\text{-}nf(M) : \bot \to \cdots \to \bot \to \tau$ *holds as well. If* $C \vdash_{(\Lambda_2^{-,*,\theta})} \theta\text{-}nf(M) : \rho$ *holds, then* $C \vdash_{(\Lambda_2^{-,*})} M : \rho$ *must hold as well.*

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

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

For convenience, we define semi-unification using the set of open types $R(0)$ as the set of algebraic terms $\mathcal{T}$. Let $X = \mathbb{V}$ denote the set of term variables to emphasize their use in algebraic terms as opposed to types. Although the members of $\mathcal{T}$ are also types, we will refer to them as

terms when using them in semi-unification. A *substitution* is a function $S : X \to \mathcal{T}$ that differs from the identity on only finitely many variables. Every substitution extends in a natural way to a $\to$-homomorphism $S : \mathcal{T} \to \mathcal{T}$ so that $S(\sigma \to \tau) = S(\sigma) \to S(\tau)$. An *instance* Γ of *semi-unification* is a finite set of pairs (called inequalities) in $\mathcal{T} \times \mathcal{T}$. Each such pair is written as $\tau \leq \mu$ where $\tau, \mu \in \mathcal{T}$. A substitution S is a *solution* of instance $\Gamma = \{\tau_1 \leq \mu_1, \ldots, \tau_n \leq \mu_n\}$ if and only if there exist substitutions $R_1, \ldots, R_n$ such that:

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

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

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

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

where:

$$\begin{aligned}
V_0 &= FV(\tau^{1,1}) \cup \cdots \cup FV(\tau^{1,r_1}) \\
V_1 &= FV(\mu^{1,1}) \cup \cdots \cup FV(\mu^{1,r_1}) \cup FV(\tau^{2,1}) \cup \cdots \cup FV(\tau^{2,r_2}) \\
&\vdots \\
V_{n-1} &= FV(\mu^{n-1,1}) \cup \cdots \cup FV(\mu^{n-1,r_{n-1}}) \cup FV(\tau^{n,1}) \cup \cdots \cup FV(\tau^{n,r_n}) \\
V_n &= FV(\mu^{n,1}) \cup \cdots \cup FV(\mu^{n,r_n})
\end{aligned}$$

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

We now define a procedure which constructs a solution for ASUP instance Γ if Γ has a solution and otherwise answers that there is no solution. This procedure is a modification of the procedure defined in [KTU93] which consists of repeatedly reducing *redexes*, which can be of two kinds, and it halts when there are no more redexes or when a conflict is detected that precludes a solution. Each reduction substitutes a term for a variable throughout Γ and the composition of the reductions done so far represents the construction of the solution.

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

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

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

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

$$\Pi(\tau) = \Delta(\tau) \in X \quad \text{and} \quad \Sigma\Pi(\mu) = \xi \quad \text{and} \quad \Sigma\Delta(\mu) = \mu'$$

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

Although the general case of semi-unification was proven to be undecidable in [KTU93], we have the following result for ASUP:

Lemma 6.1 *For an instance Γ of ASUP, the redex procedure either constructs a solution S to Γ and halts or correctly answers that Γ has no solution and halts.*

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

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

We will adopt the convention that the abstractions in the component T_i bind variables named $z_{i,1}$, $z_{i,2}$, etc. By writing the inequality $(\tau \leq_i \mu)$, we assert that the inequality will belong to column i of Γ. Most of the inequalities will be of a certain special form, so $(\tau \doteq_i \mu)$ denotes the inequality $(\alpha{\to}\alpha \leq_i \tau{\to}\mu)$ where α is a fresh variable mentioned in no other term in Γ. This will have the effect of unifying τ and μ as in ordinary first-order unification. We will assume that the subterms of M are indexed so that two otherwise identical subterms in different positions within M will be considered distinct in what follows.

We construct Γ as follows. In constructing the instance Γ of ASUP, each subterm $N \subset T_i$ will contribute one inequality, each β-redex $((\lambda^1 y_i.P_i)T_i)$ will contribute one inequality, and for each variable y_i there will be $1 + n - i$ inequalities. For each subterm N of T_i, the term variable δ_N will represent the derived type of N. For each bound variable $z_{i,j}$ (which must be monomorphic), the term variable $\gamma_{i,j}$ will represent its assigned type. For each bound variable y_i (which must be universally polymorphic), the term variables $\beta_{i,i}, \ldots, \beta_{n,i}$ will represent its assigned type. For each occurrence of x_j (which will be assigned the type $\perp$), there will be no particular variable to represent its type, since it is unconstrained.

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

1. For $N \equiv x_j$, we do not add any inequality.
2. For $N \equiv y_j$, we add $(\beta_{i-1,j} \leq_i \delta_N)$.
3. For $N \equiv z_{i,j}$, we add $(\gamma_{i,j} \doteq_i \delta_N)$.
4. For $N \equiv (PQ)$, we add $(\delta_P \doteq_i \delta_Q{\to}\delta_N)$.
5. For $N \equiv (\lambda^3 z_{i,j}.P)$, we add $(\gamma_{i,j}{\to}\delta_P \doteq_i \delta_N)$.

For each β-redex $((\lambda^1 y_i.P_i)T_i)$, we add the inequality $(\beta_{i,i} \doteq_i \delta_{T_i})$. Finally, for each bound variable y_j and for each $i \in \{j+1, \ldots, n+1\}$, we add the inequality $(\beta_{i-1,j} \leq \beta_{i,j})$.

Theorem 6.2 *The ASUP instance Γ has a solution S if and only if the λ-term M in θ-normal form is typable in $\Lambda_2^{-,*,\theta}$. Furthermore, if M is typable in $\Lambda_2^{-,*,\theta}$, the type $\perp{\to}\cdots{\to}\perp{\to}(S(\delta_{T_{n+1}}))$ where the number of "$\perp$" components of the type is m (the number of variables in the sequence x_1, $\ldots$, x_m, also the value of $|act(M)|$) is a type derivable for M in $\Lambda_2^{-,*,\theta}$.*

We can finally describe our type inference algorithm for System Λ_2. If M is typable in Λ_2, then the following procedure will produce a type for it and will otherwise answer that M is not typable:

1. Compute the labelled $M_1 = (M)^\lambda$.
2. Compute the λ-term $M_2 = \theta\text{-nf}(M_1)$ using θ-reduction.
3. Compute the ASUP instance Γ.
4. Run the redex procedure on Γ to either produce a solution S for Γ or the answer that Γ has no solution. In the latter case, halt with the answer that M is not typable in Λ_2.
5. Compute and output the type $\bot\to\cdots\to\bot\to(S(\delta_{T_{n+1}}))$ where the number of "$\bot$" components is $|act(M)|$.

It was shown in [KT92] that Λ_2 typability is DEXPTIME-complete (where DEXPTIME means $\text{DTIME}(2^{n^{O(1)}})$). We have just developed an algorithm that reduces Λ_2 type inference to ASUP in polynomial-time. ASUP was shown to be DEXPTIME-complete in [KTU90]. Therefore, our algorithm is optimal.

References

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

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

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

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

[GRDR91] P. Giannini and S. Ronchi Della Rocca. Type inference in polymorphic type discipline. In *Theoretical Aspects Comput. Softw. : Int'l Conf.*, vol. 526 of *LNCS*, pp. 18–37. Springer-Verlag, Sept. 24–27, 1991.

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

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

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

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

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

[Lei83] D. Leivant. Polymorphic type inference. In *Conf. Rec. 10th Ann. ACM Symp. Princ. Program. Lang.*, pp. 88–98. ACM, 1983.

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

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

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

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

[Rey74] J. C. Reynolds. Towards a theory of type structures. In [*Proc. 1st Int'l*] *Symp. Programming*, vol. 19 of *LNCS*, pp. 408–425. Springer-Verlag, 1974.

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

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

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

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

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