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

Abstract

We examine the problem of type inference for a family of polymorphic type systems containing the power of Core-ML. This family comprises the levels of the stratification of the second-order λ-calculus (system **F**) by "rank" of types. We show that typability is an undecidable problem at every rank $k \geq 3$. While it was already known that typability is decidable at rank 2, no direct and easy-to-implement algorithm was available. We develop a new notion of λ-term reduction and use it to prove that the problem of typability at rank 2 is reducible to the problem of acyclic semi-unification. We also describe a simple procedure for solving acyclic semi-unification. Issues related to principal types are discussed.

1 Introduction

Background and Motivation. Many modern functional programming languages use polymorphic type systems that support automatic type inference. Automatic type inference for untyped or partially typed programs saves the programmer the work of specifying the type of every identifier. Type polymorphism lets the programmer write generic functions that work uniformly on arguments of different types and it thus avoids the maintenance problem that results from duplicating similar program 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], Haskell [HW88]). ML shares with Algol 68 properties of compile-time type checking, strong typing and higher-order functions while also providing automatic type inference and type polymorphism.

The usefulness of a particular polymorphic type system depends very much on how feasible the task of type inference is. We define concepts in terms of the λ-calculus, which we use as our pure functional programming language throughout this paper. 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 deciding typability and performing type inference be done?

The answer to this question determines whether the type system is feasible to implement. Another related issue is:

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

Closely related to the issue of principal types is *type checking*, 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.

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 [GRDR91] and by limiting the number of generations of instantiation of quantifiers themselves introduced by instantiation [Lei91]. One natural restriction which we consider in this paper results from stratifying **F** according to the "rank" of types

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

LISP 94 - 6/94 Orlando, Florida USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

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 automatic type inference will be infeasible or impossible. 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 [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. 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. The Systems Λ_1 and Λ_0 are both equivalent to the simply-typed λ-calculus and are uninteresting. 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. Since this previous algorithm is a reduction to the type inference algorithm of **ML**, it can not possibly be as simple. In this paper, we give a simpler and more direct algorithm for the decidable case of typability in Λ_2 which does not depend on the type inference algorithm of **ML**. We first prove that Λ_2 is equivalent to a restriction named System $\Lambda_2^{-,*}$ 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 $\Lambda_2^{-,*}$. We define a further restricted System $\Lambda_2^{-,*,\theta}$ to take advantage of this. The type inference problem in $\Lambda_2^{-,*,\theta}$ for a λ-term in θ-normal form is easily converted into an instance of the acyclic semi-unification problem. Finally, we give a simple algorithm for solving the acyclic semi-unification problem. The complexity of the whole procedure is the same as that of type inference in **ML**.

The principal typing situation for Λ_2 is not as nice as for **ML**. For a given λ-term, there is no principal type such that every type derivable for the λ-term can be seen as a substitution instance of the type. We show there is a weak form of principal typing where the free type variables of a type can have open types substituted for them, but this does not allow a single type to generate all of the possible types for a λ-term. Quirks of the typing system that occur due to the lack of principal types are discussed.

We omit proofs of lemmas and theorems in this conference report to remain within the page limit. We postpone to either a later extended version of this paper or another paper discussion of the relationship between **ML**-typability and typability in Λ_2 and of type checking in Λ_2.

Acknowledgements. A number of definitions used in this paper were lifted from [KT92, KTU90, KTU93], so credit goes to both Tiuryn and Urzyczyn.

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.

In our presentation, we use the "Curry view" of type systems for the λ-calculus, in which 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.

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 $C[\;\;]$ 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 $C[\;\;]$, *including* the capture of free variables in M by the bound variables of $C[\;\;]$. 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. The symbol **K** denotes the standard combinator $(\lambda x.\lambda y.x)$ and the symbol **I** denotes $(\lambda x.x)$.

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 $\mathbb{T} ::= \mathbb{V} \mid (\mathbb{T} \rightarrow \mathbb{T}) \mid (\forall\mathbb{V}.\mathbb{T})$. A type is therefore either

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

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

a type variable or a $\to$-*type* or a $\forall$-*type*. 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 $\mathbb{X} \subseteq \mathrm{FTV}(\sigma)$, the shorthand notation $\forall\mathbb{X}.\sigma$ is defined so that $\forall\varnothing.\sigma = \sigma$ and $\forall(\mathbb{X} \cup \{\alpha\}).\sigma = \forall\alpha.\forall(\mathbb{X} - \{\alpha\}).\sigma$. 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 $\alpha_1, \ldots, \alpha_n$. 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 by *rank*. First define $\mathbb{R}(0)$ as the set of open types, i.e. types not mentioning the symbol "$\forall$". Then, for all $k \geq 0$, define $\mathbb{R}(k+1)$ by the grammar

$$\mathbb{R}(k+1) ::= \mathbb{R}(k) \mid (\mathbb{R}(k) \to \mathbb{R}(k+1)) \mid (\forall\mathbb{V}.\mathbb{R}(k+1))$$

We say that $\mathbb{R}(k)$ 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. Our definition of rank is exactly the same as the notion of rank introduced by Leivant [Lei83]. Since $\mathbb{R}(k) \subseteq \mathbb{R}(k+1)$ it follows that if a type σ is of rank k, then it also belongs to every rank $n \geq k$. Observe that the result of the substitution $\sigma[\alpha := \tau]$ may not belong to the same ranks to which σ belongs. The resulting rank depends on the rank of τ and how deep in the negative (left-side) scope of "$\to$" the free occurrences of α in σ are.

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

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

To define Λ_2^-, we will define the set $\mathbb{S} \subset \mathbb{T}$ of *restricted types* in which quantifiers can not occur immediately to the right of the arrow "$\to$". The set $\mathbb{S}$ is defined by the two grammar productions:

$$\mathbb{S} ::= \mathbb{S}' \mid (\forall\mathbb{V}.\mathbb{S})$$
$$\mathbb{S}' ::= \mathbb{V} \mid (\mathbb{S} \to \mathbb{S}')$$

The notation $\mathbb{S}(k)$ is defined to mean $\mathbb{S} \cap \mathbb{R}(k)$ and $\mathbb{S}'(k)$ similarly means $\mathbb{S}' \cap \mathbb{R}(k)$.

An *sequent* 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 sequent's *type* is the type $\sigma_1 \to \cdots \to \sigma_n \to \tau$ and a sequent's *rank* is the rank of its type. For example, a sequent $A \vdash M : \tau$ is of rank 2 if and only if τ is of rank 2 and all the types assigned by A are of 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 $A'(x) = \sigma[\vec{\alpha} := \vec{\chi}]$. We assume that throughout a sequent 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).

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 $\mathcal{K}$.

System **F** is the type system that can derive types for λ-terms using the inference rules presented in Figure 1 with no other restrictions. For every $k \geq 0$, System Λ_k is the restriction of **F** which allows only sequents of rank k to be derived.

Definition 2.1 (System Λ_2^-) We define System Λ_2^- as a restriction of System Λ_2 where the two differences are:

1. In Λ_2^- all sequents must have types in $\mathbb{S}(2)$. Thus, all assigned types are in $\mathbb{S}(1)$ and all derived types are in $\mathbb{S}(2)$.
2. The inference rule INST of Λ_2 is replaced by the rule INST$^-$ described in Figure 2 which forbids instantiation with polymorphic types.

To make this paper more self-contained, we will briefly describe the difference in the types that can be assigned to a λ-term in Λ_2 and Λ_2^-. For this description, let us temporarily suppose that quantifiers introduced into types by the INST rule are marked with the "#" symbol. For example, from the sequent $A \vdash M : \forall\alpha.(\alpha \to \alpha)$, we can derive using INST the sequent $A \vdash M : (\forall^{\#}\beta.\beta) \to (\forall^{\#}\beta.\beta)$. These markers on quantifiers do not affect the behavior of the inference rules; they merely allow us to precisely phrase our description.

Definition 2.2 (Quantifier Shifting) We define a mapping $(\ )^\bullet$ that maps every type τ, where quantifiers in τ might be marked with #, to a restricted type in $\mathbb{S}$. The mapping $(\ )^\bullet$ is defined inductively on the structure of types as follows:

$$\begin{aligned}(\alpha)^\bullet &= \alpha\\ (\sigma \to \tau)^\bullet &= \forall\vec{\alpha}.((\sigma)^\bullet \to \rho)\\ &\quad \text{where } (\tau)^\bullet = \forall\vec{\alpha}.\rho \text{ and } \rho \text{ is not a } \forall\text{-type}\\ (\forall\alpha.\sigma)^\bullet &= \forall\alpha.(\sigma)^\bullet\\ (\forall^{\#}\alpha.\sigma)^\bullet &= (\sigma)^\bullet\end{aligned}$$

For a type assignment A, we define $(A)^\bullet$ so that for every term variable x in the domain of A it holds that $(A)^\bullet(x) = (A(x))^\bullet$.

Theorem 2.3 (Λ_2 Has Same Power as Λ_2^-) *System Λ_2^- types the same set of λ-terms as Λ_2 with very similar types. More precisely, if the claim*

$$A \vdash_{\Lambda_2} M : \tau$$

holds, with the additional assumption that quantifiers introduced by INST are marked, then the claim

$$(A)^\bullet \vdash_{(\Lambda_2^-)} M : (\tau)^\bullet$$

holds as well. Also, every derivation in Λ_2^- is immediately a derivation in Λ_2. Thus, Λ_2 and Λ_2^- type the same set of λ-terms.

Theorem 2.3 is Theorem 9 in [KT92]. 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.

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

In this section, we describe a family of type systems, $\Lambda_k[C_k]$ for each $k \geq 3$, for which typability has already been shown to be undecidable. Then we show that the typability problem for each member $\Lambda_k[C_k]$ in this family of type systems is reducible to the typability problem for the corresponding type system Λ_k, thus proving it undecidable as well.

Section 5 of [KT92] introduces System $\Lambda_k[C_k]$ for each $k \geq 3$. 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 $\phi_{f,k}$ which depend on k. The types $\phi_{c,k}$ and $\phi_{f,k}$ are defined by a simple induction. Let the type $\tau_0 = \alpha$ and then let $\tau_{k+1} = (\tau_k \to \alpha)$ for $k \geq 0$. Then define $\phi_{c,k} = \forall.(\alpha \to \tau_k)$ and $\phi_{f,k} = \forall.((\alpha \to \alpha) \to \tau_{k-1})$. (The fact that both of the types $\phi_{c,k}$ and $\phi_{f,k}$ belong to $\mathbb{S}(1)$ for every k is irrelevant to the definition of System $\Lambda_k[C_k]$.) As an example, for $k = 3$, the types are

$$\begin{aligned}\phi_{c,3} &= \forall\alpha.(\alpha \to (((\alpha \to \alpha) \to \alpha) \to \alpha))\\ \phi_{f,3} &= \forall\alpha.((\alpha \to \alpha) \to ((\alpha \to \alpha) \to \alpha))\end{aligned}$$

A simple alternate definition of $\Lambda_k[C_k]$ which we use in this paper 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.

Lemma 3.1 ($\Lambda_k[C_k]$ Reducible to Λ_k) *For each $k \geq 3$, the problem of typability in the system $\Lambda_k[C_k]$ is reducible to the problem of typability in the system Λ_k. More precisely, for each $k \geq 3$, there is a context $H_k[\ \ ]$ with one hole such that for any type assignment A, the statement:*

$$\exists\tau \in \mathbb{R}(k) \text{ such that } A \cup \{c : \phi_{c,k}, f : \phi_{f,k}\} \vdash_{\Lambda_k} M : \tau$$

is true if and only if the following statement is true:

$$\exists\sigma \in \mathbb{R}(k) \text{ such that } A \vdash_{\Lambda_k} H_k[M] : \sigma$$

As an example, the context $H_3[\ \ ]$ with one hole may be constructed as depicted in Figure 3. It can be easily checked that the context in Figure 3 can be typed in Λ_3 and somewhat more tediously checked that 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 $\phi_{f,3}$. The methods of [Wel93] may be used in a similar manner to construct contexts $H_4[\ \ ]$, $H_5[\ \ ]$, $H_6[\ \ ]$, etc., each more complicated than the previous one.

Theorem 3.2 (Rank $\geq$ 3 Typability Undecidable) *For $k \geq 3$, since the problem of typability for $\Lambda_k[C_k]$ is reducible to the same problem for Λ_k, and since typability for $\Lambda_k[C_k]$ is undecidable, it is the case that typability is undecidable for Λ_k.*

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 (Active Abstractions) Define, by induction on λ-terms M, the sequence $act(M)$ of *active abstractions* in M:

$$\begin{aligned}act(x) &= \varepsilon \text{ (the empty sequence)}\\ act(\lambda x.M) &= x \cdot act(M)\\ act(MN) &= \begin{cases}\varepsilon & \text{if } act(M) = \varepsilon,\\ x_2 \cdots x_n & \text{if } act(M) = x_1 \cdots x_n \text{ for } n \geq 1.\end{cases}\end{aligned}$$

Observe that, due to our conventions on the naming of bound variables, 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.

Definition 4.2 (Companions) 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 N is R itself). In this case, we say that the subterms N and S are *companions*. Specifically, N is the *companion abstraction* and S the *companion argument*. If $N \equiv R$, i.e. $Q \equiv NS$, then we say that they are *adjacent companions*.

$$
\begin{aligned}
J_\imath[\ \ ] &\equiv (\lambda y_\imath.(\lambda z_\imath.r(y_\imath y_\imath(y_\imath z_\imath))))(\lambda x_\imath.\mathsf{K}x_\imath(\mathsf{K}(x_\imath(x_\imath r))[\ \ ]))(\lambda w_\imath.w_\imath w_\imath)\\
D[\ \ ] &\equiv (\lambda f.r(x_1(fx_1x_1))(x_2(fx_2x_2))[\ \ ])(\lambda u.\lambda v.u(v(u(ur))))\\
E[\ \ ] &\equiv (\lambda t.r(x_1(tx_1(x_1r)(fx_1)))(x_2(tx_2(x_2r)(fx_2)))[\ \ ])(\lambda p.\lambda q.\lambda s.\mathsf{K}(p(pq))(p(sp)))\\
G[\ \ ] &\equiv (\lambda c.r(x_1(c(x_1r)(fx_1)))(x_2(c(x_2r)(fx_2)))[\ \ ])(tr)\\
H_3[\ \ ] &\equiv \lambda r.J_1[J_2[D[E[G[\ \ ]]]]]
\end{aligned}
$$

Figure 3: The Context $H_3[\ ]$ used in Reduction from $\Lambda_3[C_3]$ to Λ_3.

It is the case that adjacent companions are always 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. Consider a λ-term M with subterms $(\lambda x.P)$ and Q which are companions where $(\lambda x.P)$ is the companion abstraction and Q is the companion argument. In this case, if $(\lambda x.P)$ ever participates in a β-redex after some number of steps of β-reduction, its argument will be Q or Q's β-descendent.

Companions will turn out to have convenient properties in System Λ_2^-.

Definition 4.3 (Abstraction Labelling) 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 term variables, and an index. The inductive definition of *label* follows for $i \in \{1,2,3\}$:

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

We then finish the definition by specifying that

$$(M)^\lambda = label(M, act(M), 2)$$

Informally, labeling the λ-term M affects each abstraction subterm $(\lambda x.N)$ as follows. If $(\lambda x.N)$ has a companion within M, then it is labelled as $(\lambda^1 x.N)$. If $(\lambda x.N)$ does not have a companion within M and if there is no subterm $P = LR$ of M such that N lies within R (the right subterm), then it is labelled as $(\lambda^2 x.N)$. Otherwise it is labelled as $(\lambda^3 x.N)$.

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 4.4 (λ^3 Binds Monomorphically) *The bound variable of a companionless, λ^3-labelled abstraction must be assigned a monomorphic type. More precisely, 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 $act(Q)$ (i.e. labelling would produce $(\lambda^3 x.N)$), then for every sequent $A \cup \{x:\sigma\} \vdash N : \tau$ in $\mathcal{D}$ it is the case that $\sigma \in \mathbb{S}(0)$.*

Lemma 4.5 (Free Type Variable Substitution) *If $\mathcal{D}$ is a derivation in Λ_2^- ending with the sequent $A \vdash M : \tau$, then for any type variable substitution $[\vec{\alpha} := \vec{\chi}]$, it is the case that there is a derivation $\mathcal{D}'$ in Λ_2^- ending with the sequent $A[\vec{\alpha} := \vec{\chi}] \vdash M : \tau[\vec{\alpha} := \vec{\chi}]$ 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 4.5 is used by Lemma 4.6. For Lemma 4.6, let us temporarily suppose that quantifiers introduced into types by the GEN rule are marked with the "$\flat$" symbol. For example, from the sequent $A \vdash M : \tau$ where $\alpha \notin \mathrm{FTV}(A)$ we can derive using GEN the sequent $A \vdash M : \forall^\flat \alpha.\tau$. These markers on quantifiers do not affect the behavior of the inference rules; they merely allow us to precisely phrase the claim of the lemma.

Lemma 4.6 (GEN Quantifiers Not Instantiated) *We may freely assume that quantifiers introduced by* GEN *are never instantiated. More precisely, if $\mathcal{D}$ is a derivation in Λ_2^- ending with the sequent $A \vdash M : \tau$, then there is a derivation $\mathcal{D}'$ in Λ_2^- ending with the same sequent such that in $\mathcal{D}'$ there is no use of the* INST *rule whose premise is a sequent of the form $B \vdash N : \forall^\flat \alpha.\rho$.*

Lemma 4.7 (Outermost Quantifiers Only at Companion Arguments) *The only proper subterms of a λ-term for which the final derived type may be a $\forall$-type are companion arguments. More precisely, if $\mathcal{D}$ is a derivation in Λ_2^- that types the λ-term M, and if $\mathcal{D}$ includes the sequent $A \vdash N : \forall\alpha.\tau$, and if there are no subsequent sequents in $\mathcal{D}$ for the same occurrence of the subterm N, then either $N \equiv M$ or this occurrence of N is the argument subterm of a subterm (PN) in M where $act(P) \not\equiv \varepsilon$.*

Lemma 4.8 results from Lemmas 4.6 and 4.7.

Lemma 4.8 (GEN Only at Companion Arguments) *The only proper subterms of a λ-term for which the* GEN *rule may be used are companion arguments. More precisely, if $\mathcal{D}$ is a derivation in Λ_2^- that types the λ-term M, and if $\mathcal{D}$ includes the sequent $A \vdash N : \forall\alpha.\tau$ as a consequence of the* GEN *rule, and if $N \not\equiv M$, then N is a companion argument.*

Lemma 4.9 (INST Only at Variables) *We may freely assume that all uses of the* INST *rule occur at the leaves of the derivation (viewing the derivation as a tree). More precisely, if $\mathcal{D}$ is a derivation in Λ_2^- ending with the sequent $A \vdash M : \tau$, then there is a derivation $\mathcal{D}'$ in Λ_2^- ending with the same sequent such that if the sequent $B \vdash N : \sigma$ in $\mathcal{D}'$ is the consequence of the* INST *rule, then N is a term variable.*

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

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

Definition 4.10 (System $\Lambda_2^{-,*}$) The new System $\Lambda_2^{-,*}$ formally includes the restrictions on Λ_2^- proven by the previous lemmas in a type system. The inference rules for $\Lambda_2^{-,*}$ are in Figure 4. As in Λ_2^-, all sequents are required to be of rank 2, i.e. assigned types must be in $\mathbb{S}(1)$ and derived types must be in $\mathbb{S}(2)$.

Theorem 4.11 ($\Lambda_2^{-,*}$ Equivalent to Λ_2^-) *Every Λ_2^- typing is equivalent to a $\Lambda_2^{-,*}$ typing and vice versa. More precisely, the claim:*

$$A \vdash_{(\Lambda_2^-)} M : \tau$$

holds if and only if the following claim holds:

$$A \vdash_{(\Lambda_2^{-,*})} (M)^\lambda : \tau$$

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-Reduction) 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 $(\lambda^3 x.(\lambda^1 y.N)P)$ into the form $((\lambda^1 v.\lambda^3 x.(N[y{:=}vx]))(\lambda^3 w.(P[x{:=}w])))$, 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 $\theta = \theta_1 \cup \theta_2 \cup \theta_3 \cup \theta_4$. Since these transformations are all notions of reduction, the notations $\to_{\theta_1}$, $\to_{\theta_2}$, $\to_\theta$, 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: the λ-term $((\lambda x.(\lambda y.MP)Q)N)$ and the λ-term $((\lambda y.(\lambda x.MP)N)Q)$.

We now describe some useful properties of θ-reduction.

Lemma 5.2 (Shape of θ-Normal Forms) *Let M be in θ-normal form. Then M is of the form*

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

where $m, n \geq 0$ and where for $1 \leq i \leq n+1$ each subterm T_i is in β-normal form and any abtractions within T_i are λ^3-labelled.

Observe that in a θ-normal form all λ^1-labelled abstractions belong to β-redexes, i.e. there are no-adjacent companions. The λ-term M detailed in Lemma 5.2 can also be viewed as the following ML term:

fn $x_1 \Rightarrow$ **fn** $x_2 \Rightarrow \cdots \Rightarrow$ **fn** $x_m \Rightarrow$
let $y_1 = T_1$ **in let** $y_2 = T_2$ **in** $\cdots$
let $y_n = T_n$ **in** T_{n+1}

Lemma 5.3 (β-Equivalence Preserved) *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 $M \to_\theta N$, then $M =_\beta N$.*

To prove that any θ-reduction terminates, we establish a metric on λ-terms and we then show that θ-reduction strictly decreases this metric.

Definition 5.4 (Distance from θ-Normal Form) We will define a function from labelled λ-terms to natural numbers using the following components. In the following definitions, we presume that each subterm of a λ-term is somehow distinctly indexed, so that otherwise identical subterms in different positions are distinguished. This is important so that the desired answers are produced when counting the size of a set of subterms and when asking whether one subterm is a subterm of another subterm.

Let A (for "ancestors") be a function that takes a labelled λ-term M and a subterm N within M, and returns the set of all subterms of M which contain N (including M and N). Let β (for "β-redexes") be a function that takes a λ-term M and returns the set of all of the subterms of M that are either β-redexes or are the function of a β-redex. Let λ^i for each $i \in \{1,2,3\}$ be a function that takes a λ-term M and returns the set of all subterms of M that are λ^i-labelled abstractions.

Now define three metric functions δ_1, δ_2, and δ_3 which are used to measure the distance of a λ-term from θ-normal form. δ_1 takes a λ-term M and a subterm N and returns the number of subterms of M that contain N that are neither β-redexes, the function of a β-redex, nor a λ^2-labelled abstraction:

$$\delta_1(M,N) = \left| A(M,N) - \beta(M) - \lambda^2(M) \right|$$

δ_2 takes a λ-term M and a subterm N and returns the number of application subterms in M that contain N as a subterm of their right subterm:

$$\delta_2(M,N) = \left| \{ P \mid P \in A(M,N),\ P \equiv QR,\ R \in A(M,N) \} \right|$$

δ_3 takes a λ-term M and a subterm N and returns the number of subterms in M that are λ^2-labelled abstraction and do not properly contain N:

$$\delta_3(M,N) = \left| \lambda^2(M) - \{N\} - A(M,N) \right|$$

Now use δ_1, δ_2, and δ_3 to define the metric function d to measure how far a λ-term is from θ-normal form. Define d as follows:

$$d(M) = \sum_{N \in \lambda^1(M)} \delta_1(M,N) + \delta_2(M,N) + \sum_{N \subset M} \delta_3(M,N)$$

Note that δ_1 and δ_2 are applied just to the λ^1-labelled abstractions in M, i.e. the abstractions that have companions, but δ_3 is applied to all subterms of M.

Lemma 5.5 (θ-Reduction Terminates) *θ-reduction always terminates (strongly normalizes). More precisely, if $M \rightarrow_{\theta_i} N$, then $d(M) > d(N)$. Furthermore, for a λ-term M, it holds that $d(M) \in O(|M|^2)$, so it takes $O(|M|^2)$ steps of θ-reduction to reach θ-normal form.*

Lemma 5.6 (β-Redex Binds Partly Closed Type) *We may freely assume that for the type σ assigned to the bound variable of a λ^1-abstraction which is the function of a β-redex, it is the case that any free type variables in σ must also be free somewhere else in the type assignment. More precisely, if $\mathcal{D}$ is a derivation in $\Lambda_2^{-,*}$ containing the sequent $A \vdash (\lambda^1 x.M)N : \tau$ which is derived from the earlier sequents $A \cup \{x : \sigma\} \vdash M : \tau$ and $A \vdash N : \sigma$, then there is also a derivation $\mathcal{D}'$ in $\Lambda_2^{-,*}$ containing the same sequent but in which the sequent is derived instead from earlier sequents $A \cup \{x : \sigma'\} \vdash M : \tau$ and $A \vdash N : \sigma'$ where $\sigma' \equiv \forall(\mathrm{FTV}(\sigma) - \mathrm{FTV}(A)).\sigma$.*

Lemma 5.6 is used by Lemma 5.7.

Lemma 5.7 ($\Lambda_2^{-,*}$ Typings Preserved) *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 $A \vdash M : \tau$ is derivable if and only if $A \vdash N : \tau$ 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 5.8 (Active Abstractions Preserved) *The set of active abstractions of a λ-term is preserved by θ-reduction. As a result, $act(\theta\text{-}nf((M)^\lambda)) = act(M)$.*

Lemma 5.9 (Shape of Derivable Types) *In $\Lambda_2^{-,*}$, if $A \vdash M : \rho$ is derivable and $|act(M)| = n$, then*

$$\rho = \forall\vec{\alpha}.\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \tau$$

where $\vec{\sigma} \in \mathbb{S}(1)$ and $\tau \in \mathbb{S}(0)$.

Lemma 5.9 was proven in [KT92].

Lemma 5.10 (λ^2 Can Bind Closed Type) *We can always assign a closed type or even 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. More precisely, if $\mathcal{D}$ is a typing in $\Lambda_2^{-,*}$ of the λ-term M ending with the sequent*

$$A \vdash M : \forall\vec{\alpha}.\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \tau$$

where $|act(M)| = n$, then there is a typing $\mathcal{D}'$ ending with the sequent

$$A \vdash M : \forall\vec{\beta}.(\forall.\sigma_1) \rightarrow \ldots \rightarrow (\forall.\sigma_n) \rightarrow \tau$$

where $\vec{\beta} = \vec{\alpha} - (\mathrm{FTV}(\vec{\sigma}) - \mathrm{FTV}(\tau))$ and there is also a derivation $\mathcal{D}''$ ending with the sequent

$$A \vdash M : \forall\vec{\beta}.\bot \rightarrow \ldots \rightarrow \bot \rightarrow \tau$$

Lemma 5.11 (λ^1 Can Bind Closed Type in θ-nf) *Provided the final type assignment in a derivation assigns closed types to all free variables, and provided that every λ^2-abstraction binds a variable with a closed type, and provided we are typing a λ-term in θ-normal form, then we can assign closed types to the bound variables of every λ^1-labelled abstraction without affecting the whole λ-term's typability.*

Definition 5.12 (System $\Lambda_2^{-,*,\theta}$) The new System $\Lambda_2^{-,*,\theta}$ takes 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 5. As for $\Lambda_2^{-,*}$, all sequents are required to be of rank 2, i.e. assigned types must be in $\mathbb{S}(1)$ and derived types must be in $\mathbb{S}(2)$. We adopt the convention that the final type assignment of any typing in $\Lambda_2^{-,*,\theta}$ must assign closed (universally polymorphic) types to every free variable, otherwise the derivation is considered incomplete.

VAR^{θ}	$A \vdash x : \tau$	$A(x) \preceq^0 \tau, \quad \tau \in \mathbb{S}(0)$
APP^{θ}	$\dfrac{A \vdash M : \sigma \to \tau, \qquad A \vdash N : \sigma}{A \vdash (M\,N) : \tau}$	$\sigma, \tau \in \mathbb{S}(0), \quad M$ not abstraction
LET^{θ}	$\dfrac{A \cup \{x : \forall.\sigma\} \vdash M : \tau, \qquad A \vdash N : \sigma}{A \vdash ((\lambda^1 x.M)\,N) : \tau}$	$\sigma, \tau \in \mathbb{S}(0), \quad \mathrm{FTV}(A) = \varnothing$
$\mathrm{ABS}^{\theta,2}$	$\dfrac{A \cup \{x : \forall.\sigma\} \vdash M : \tau}{A \vdash (\lambda^2 x.M) : (\forall.\sigma) \to \tau}$	$\sigma \in \mathbb{S}(0), \quad \tau \in \mathbb{S}'(2)$
$\mathrm{ABS}^{\theta,3}$	$\dfrac{A \cup \{x : \sigma\} \vdash M : \tau}{A \vdash (\lambda^3 x.M) : \sigma \to \tau}$	$\sigma, \tau \in \mathbb{S}(0)$

Figure 5: Inference Rules of System $\Lambda_2^{-,*,\theta}$.

Theorem 5.13 **($\Lambda_2^{-,*}$ Reducible to $\Lambda_2^{-,*,\theta}$)** *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*

$$A \vdash_{(\Lambda_2^{-,*})} M : \forall\vec{\alpha}.\sigma_1 \to \cdots \to \sigma_n \to \tau$$

is true, then using the type assignment B such that for $x \in \mathrm{FV}(M)$ it behaves so that $B(x) = \forall.A(x)$, it is the case that

$$B \vdash_{(\Lambda_2^{-,*,\theta})} \theta\text{-}nf(M) : (\forall.\sigma_1) \to \cdots \to (\forall.\sigma_n) \to \tau$$

and using the type assignment C that maps all free variables to type $\bot$ it is the case that

$$C \vdash_{(\Lambda_2^{-,*,\theta})} \theta\text{-}nf(M) : \bot \to \cdots \to \bot \to \tau$$

Also, every derivation in $\Lambda_2^{-,*,\theta}$ is immediately a derivation in $\Lambda_2^{-,*}$, so if

$$A \vdash_{(\Lambda_2^{-,*,\theta})} \theta\text{-}nf(M) : \rho$$

is true, then

$$A \vdash_{(\Lambda_2^{-,*})} M : \rho$$

must be true as well.

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

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

Definition 6.1 (Semi-Unification (SUP)) For convenience, we define semi-unification using the set of open types $\mathbb{R}(0)$ 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:

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

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

Definition 6.2 (Acyclic Semi-Unification (ASUP)) 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:

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

where for $0 \leq i \leq n$:

$$V_i = \{\, \alpha \mid \exists j.\, \alpha \in \mathrm{FTV}(\tau^{i+1,j}) \text{ or } \alpha \in \mathrm{FTV}(\mu^{i,j}) \,\}$$

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

Definition 6.3 (Paths in Terms) For an arbitrary algebraic 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.

The following algorithm is an important sub-algorithm of the overall type-inference algorithm for Λ_2.

Algorithm 6.4 (Redex Procedure) We now define a procedure (modified from [KTU93]) to solve instances of ASUP. This procedure repeatedly reduces *redexes* of two kinds and it halts if there are no more redexes or if 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:

$$\begin{array}{rclrcl} \Pi(\tau) & \in & X & \Pi(\tau) & = & \Delta(\tau) \\ \Sigma\Pi(\mu) & = & \xi & \Sigma\Delta(\mu) & = & \mu' \end{array}$$

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 SUP has been proven to be undecidable [KTU93], ASUP has been proven to be decidable and in fact it is DEXPTIME-complete [KTU90] (where DEXPTIME means DTIME($2^{n^{O(1)}}$)). In addition, we have the following result for ASUP.

Lemma 6.5 (Redex Procedure Solves ASUP) *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. Furthermore, it halts within exponential time.*

We now define another important sub-algorithm of the type-inference algorithm for Λ_2.

Algorithm 6.6 (Constructing Γ_M) To solve the typability and type inference problems for $\Lambda_2^{-,*,\theta}$ for λ-terms in θ-normal form, we construct for a λ-term M an ASUP instance Γ_M. Consider the labelled λ-term M in θ-normal form:

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

We will adopt the convention that the abstractions in a component T_i for some i bind variables named $z_{i,1}$, $z_{i,2}$, etc., and that the free variables of M are named $w_1, w_2, \ldots, w_p$. By writing the inequality $(\tau \leq_i \mu)$, we assert that the inequality will belong to column i of Γ, which will have $n+2$ columns numbered from 0 through $n+1$. (We omit the proof that the resulting set of inequalities Γ_M is of the correct acyclic form to be an instance of ASUP.) 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 the instance Γ_M of ASUP from the λ-term M as follows. In constructing Γ_M, each subterm $N \subset T_i$ for some i will contribute one inequality, each β-redex $((\lambda^1 y_i.P_i)T_i)$ will contribute one inequality, each variable y_i will contribute $n - i + 1$ inequalities, and each variable x_i or w_i will contribute n inequalities. For each subterm N of T_i for some 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 variable x_i (respectively y_i or w_i), which must be assigned a universally polymorphic type, the term variables $\beta^x_{0,i}, \ldots, \beta^x_{n,i}$ (respectively $\beta^y_{i,i}, \ldots, \beta^y_{n,i}$ and $\beta^w_{0,i}, \ldots, \beta^w_{n,i}$) will represent its assigned type.

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

1. For $N \equiv w_j$, we add $(\beta^w_{i-1,j} \leq_i \delta_N)$.
2. For $N \equiv x_j$, we add $(\beta^x_{i-1,j} \leq_i \delta_N)$.
3. For $N \equiv y_j$, we add $(\beta^y_{i-1,j} \leq_i \delta_N)$.
4. For $N \equiv z_{i,j}$, we add $(\gamma_{i,j} \doteq_i \delta_N)$.
5. For $N \equiv (PQ)$, we add $(\delta_P \doteq_i \delta_Q \to \delta_N)$.
6. 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 $(\beta^y_{i,i} \doteq_i \delta_{T_i})$. For each variable x_j (respectively y_j or w_j) and for $1 \leq i \leq n$ (for y_j require $i \geq j+1$ as well) we add the inequality $(\beta^x_{i-1,j} \leq_i \beta^x_{i,j})$ (respectively $(\beta^y_{i-1,j} \leq_i \beta^y_{i,j})$ or $(\beta^w_{i-1,j} \leq_i \beta^w_{i,j})$).

The only remaining consideration is what types to assign to the λ^2-bound variables $x_1, \ldots, x_m$ and to the free variables $w_1, \ldots, w_p$. If our only concern is whether M can be typed at all, then we can assign the type $\bot$ to these variables, in which case we do not need to add anything more to Γ_M. On the other hand, we may wish to specify more complex assigned types for these variables. Let A be a type assignment whose domain is $\{x_1, \ldots, x_m, w_1, \ldots, w_p\}$ and whose range is in $(\mathbb{S}(1) - \mathbb{S}(0))$. We define $\Gamma_{M,A}$ to be Γ_M with the addition of more inequalities. If $A(x_i)$ (respectively $A(w_i)$) is $\forall.\sigma$ where $\sigma \in \mathbb{S}(0)$, we add $(\beta^x_{0,i} \doteq_0 \sigma)$ (respectively $(\beta^w_{0,i} \doteq_0 \sigma)$).

Theorem 6.7 ($\Lambda_2^{-,*,\theta}$ Reducible to ASUP) *Type inference in $\Lambda_2^{-,*,\theta}$ is reducible to* ASUP. *More precisely, let M be a λ-term in θ-normal form of the shape mentioned in Algorithm 6.6. Let $act(M) = x_1 \ldots x_m$. Let A be a type assignment whose domain is* $\mathrm{FV}(M) \cup \{x_1, \ldots, x_m\}$ *and whose range is in $(\mathbb{S}(1) - \mathbb{S}(0))$. Let Γ_M be the* ASUP *instance defined by the algorithm. Let $\delta_{T_{n+1}}$ be the term variable appearing in Γ_M which is mentioned in the algorithm. The following statements are true:*

1. *Γ_M has a solution S if and only if M is typable in $\Lambda_2^{-,*,\theta}$. Furthermore, if τ is the type*

$$\tau = \bot \to \cdots \to \bot \to (S(\delta_{T_{n+1}}))$$

where the number of "$\bot$" components in τ is m, then τ is a type derivable for M in $\Lambda_2^{-,*,\theta}$.

2. *If S be a solution for $\Gamma_{M,A}$, then the type*

$$\tau = A(x_1) \to \cdots \to A(x_m) \to (S(\delta_{T_{n+1}}))$$

is a type derivable for M in $\Lambda_2^{-,*,\theta}$.

Algorithm 6.8 (Type Inference for Λ_2) We can finally summarize 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 in Λ_2:

1. Compute the labelled λ-term $M_1 \equiv (M)^\lambda$.
2. Compute the λ-term $M_2 \equiv \theta\text{-nf}(M_1)$ using θ-reduction.
3. Choose a type assignment A for the free and λ^2-bound variables of M. If $act(M) = x_1 \ldots x_m$, let the domain of A be $\text{FV}(M) \cup \{x_1, \ldots, x_m\}$ and let the range of A be in $(\mathbb{S}(1) - \mathbb{S}(0))$. It is possible to choose the trivial type assignment that assigns $\bot$ to all variables.
4. Compute the ASUP instance $\Gamma_{M_2,A}$ (Algorithm 6.6).
5. Run the redex procedure (Algorithm 6.4) on $\Gamma_{M_2,A}$ to either produce a solution S for $\Gamma_{M_2,A}$ or the answer that $\Gamma_{M_2,A}$ has no solution. In the latter case, halt with the answer that M is not typable in Λ_2 with the assumptions of the type assignment A. If A is the trivial type assignment, then M is not typable at all in Λ_2.
6. Compute and output the type

$$A(x_1) \to \cdots \to A(x_m) \to (S(\delta_{T_{n+1}}))$$

where $\delta_{T_{n+1}}$ is the term variable appearing in Γ_M which is mentioned in Algorithm 6.4.

The reader should observe that Algorithm 6.8 makes no reference to the type systems Λ_2^-, $\Lambda_2^{-,*}$, or $\Lambda_2^{-,*,\theta}$. These type systems are used solely to prove that the output of the algorithm in its final step is a correct result. The final result is a valid typing in $\Lambda_2^{-,*,\theta}$, but it is also immediately a valid typing in $\Lambda_2^{-,*}$, Λ_2^-, and Λ_2 as well (after removing any λ-labelling).

We now analyze the complexity of Algorithm 6.8. The initial stages of computing the labelling $M_1 \equiv (M)^\lambda$, the θ-normal form $M_2 \equiv \theta\text{-nf}(M_1)$, and the ASUP instance $\Gamma_{M_2,A}$ can be done in polynomial time. Algorithm 6.4 solves the ASUP instance $\Gamma_{M_2,A}$ in exponential time. Thus, Algorithm 6.8 takes exponential time. Since Λ_2 typability has been shown to be DEXPTIME-complete [KT92], the algorithm is optimal.

To use System Λ_2 or Λ_2^- in an actual programming language, we will have to take account of constants with constant types, e.g. "**true** : **Bool**". This might seem difficult to do, since the type inference algorithm is based on System $\Lambda_2^{-,*,\theta}$ which requires all types assigned to identifiers to be completely closed (polymorphic). However, the redex procedure for solving ASUP instances can be simply told that certain variables are actually constants (e.g. **Bool**) and not to be changed by substitution. Then the type inference algorithm will work correctly with constants.

7 Principal Typing in System Λ_2

In this section, we first observe that in general there are no principal types for Λ_2. Then we describe the principality of solutions to instances of SUP and ASUP and how this relates to types in Λ_2. Finally, we discuss the weak forms of type principality that exist in Λ_2.

It is easy to observe that principal types do not exist in System Λ_2 in the same sense that they do in **ML**. Consider the identity function, $\mathsf{I} \equiv (\lambda x.x)$. In Λ_2^-, all of the types

$$\begin{aligned} \varphi &= (\forall\alpha.\alpha) \to (\beta \to \beta) \\ \psi &= (\forall\alpha.(\alpha \to \alpha)) \to (\beta \to \beta) \\ \pi &= \forall\alpha.(\alpha \to \alpha) \end{aligned}$$

can be derived for I. (Note that π can not be derived for I in $\Lambda_2^{-,*,\theta}$.) However, there is no type τ derivable for I in Λ_2 such that $\tau \preceq \varphi$, $\tau \preceq \psi$, and $\tau \preceq \pi$. When we consider the full power of Λ_2 and the polymorphic instantiation and types in $(\mathbb{R}(2) - \mathbb{S}(2))$ that it allows, the situation seems even more disconcerting.

We do not currently know of a convenient way to represent all of the possible rank-2 types that can be derived for a λ-term. The types derived by our type inference algorithm are principal in a weak sense. The rest of this section will present what is known about the kind of weak principality of types that exists.

The solutions to instances of SUP and ASUP are principal in a weak sense. For substitutions $S, R : X \to \mathcal{T}$, let the notation $S \sqsubseteq R$ mean that there exists some substitution $S' : X \to \mathcal{T}$ such that for all term variables α in the domain of S it holds that $R(\alpha) = S'(S(\alpha))$.

Lemma 7.1 (Principal SUP Solution) *If Γ is an instance of* SUP*, then Γ has a principal solution S such that for every solution R of Γ it is the case that $S \sqsubseteq R$.*

Lemma 7.1 is Proposition 3 in [KTU93].

Lemma 7.2 (Principal ASUP Solution) *Suppose Γ is an instance of* ASUP *with n columns. There are therefore $n+1$ disjoint sets of variables occurring in Γ, which we call V_0, V_1, ..., V_n, satisfying the property that for every inequality $(\tau \leq \mu)$, if $\alpha \in V_i$ occurs in τ, then all the term variables in τ also belong to V_i and all of the term variables in μ belong to V_{i+1}. Let $V = V_0 \cup \cdots \cup V_n$. For a substitution $T : V \to \mathcal{T}$, let the notation $[T]_i$ denote the restriction of T to the domain of V_i. Suppose S is the principal solution of Γ according to Lemma 7.1. Then the conclusion of this lemma is that for any substitution $P : V_i \to \mathcal{T}$ such that $[S]_i \sqsubseteq P$, there is a substitution $R : V \to \mathcal{T}$ such that:*

1. *R is a solution of Γ.*
2. *$[R]_i = P$.*

Now, from Lemma 7.2 follows the weak principal typing property for System $\Lambda_2^{-,*,\theta}$.

Theorem 7.3 (Weak Principal Types) *Consider the type computed by Algorithm 6.8 from $\Gamma_{M,A}$ for λ-term M relative to type assignment A for the free and λ^2-bound variables of M:*

$$A(x_1) \to \cdots \to A(x_m) \to (S(\delta_{T_{n+1}}))$$

For any substitution $P : X \to \mathcal{T}$, the following type is derivable for M relative to A:

$$A(x_1) \to \cdots \to A(x_m) \to (P(S(\delta_{T_{n+1}})))$$

Theorem 7.3 holds since S is a solution for $\Gamma_{M,A}$ and $S \sqsubseteq S \circ P$, so there is a solution R such that $[R]_{n+1} = S \circ P$ (where V_{n+1} is the rightmost set of variables in $\Gamma_{M,A}$).

Now consider the types inferred by Algorithm 6.8 for a λ-term M under various restrictions. Suppose M has no λ^2-labelled abstractions and no free variables. In that case, the computed type is exactly $S(\delta_{T_{n+1}})$. By Theorem 7.3, any substitution instance of this type is also a valid type for M. Thus, in this case there is the same sort of strong principality of types that there is in **ML**.

Now consider various cases where M has either λ^2-bound variables or free variables or both.

Suppose in type inference we decide to assign the type $\bot$ to all free and λ^2-bound variables, which provides the maximum possibilities for M to be typed. In this case, the final sequent of the typing will look like this:

$$\{w_1 : \bot, \ldots, w_p : \bot\} \vdash M : \bot \to \cdots \to \bot \to (S(\delta_{T_{n+1}}))$$

The rightmost component of the type, $S(\delta_{T_{n+1}})$, can be replaced with any substitution instance of it. However, since there are no closed λ-terms in Λ_2 for which the type $\bot$ can be derived, this typing does not help us know with what other λ-terms M can be combined. Although assigning $\bot$ to all free and λ^2-bound variables allows us to tell whether a λ-term is typable at all, it seems unlikely to be useful in practice.

A problem with the type inference algorithm and $\Lambda_2^{-,*,\theta}$ is that certain important and natural types will not be assigned to λ-terms unless a trick is used. For example, the type inference algorithm will not derive the type $\forall\alpha.(\alpha \to \alpha)$ for the λ-term $\mathbf{I} \equiv (\lambda x.x)$. The reason for this is that after labelling, the λ-term is $(\lambda^2 x.x)$ and the type assigned to λ^2-bound variables is required to be closed. The type inference algorithm can assign the type $\forall\alpha.(\alpha \to \alpha)$ to the λ-term $(\lambda^3 x.x)$, which is the same λ-term labelled differently. This is not a problem in typing a real program, because whenever the type $\forall\alpha.(\alpha \to \alpha)$ is needed for $(\lambda x.x)$, it will be the case that $(\lambda x.x)$ is embedded inside a larger term in a position where the abstraction will be λ^3-labelled. If it is desired to know what type will be assigned to the λ-term M in such a position, the type inference algorithm can be asked to type $(\mathbf{I}M)$ instead. The primary problem with this typing quirk is that the type derived for a free-standing λ-term does not indicate to the human viewer the actual possible types the λ-term can take in combination with other λ-terms. The type inference algorithm must pick some type, but, due to the lack of principal types, the type it picks can not be a most general type.

It may be desired to know the most general open type that can be assigned to a λ-term M, ignoring all of the possible rank-2 but not rank-1 final types. (This is different from **ML** typing in that rank-2 types are allowed in intermediate steps in the type derivation.) This is quite simple to do: simply ask the type inference algorithm the type of $(\mathbf{I}M)$. Any rank-1 type derivable for M in Λ_2^- is also derivable for $(\mathbf{I}M)$, but no rank-2 but not rank-1 types are derivable for $(\mathbf{I}M)$.

Similarly, it may also be desired to find a most general typing in which all types in the final sequent are open. To find such a typing for λ-term M with free variables $w_1, \ldots, w_n$, use the type inference algorithm to compute the type for the λ-term $(\mathbf{I}(\lambda w_1.\ldots.\lambda w_n.M))$, which will be of the shape $\rho_1 \to \cdots \to \rho_n \to \varphi$, where $\vec{\rho}, \varphi \in \mathbb{S}(0)$. In this case, any type-substitution instance of the following sequent will be derivable:

$$\{w_1 : \rho_1, \ldots, w_n : \rho_n\} \vdash M : \varphi$$

It may be desired to use specific closed types for some of the free or λ^2-bound variables of a λ-term, but to have the type inference algorithm compute most general open types for the rest of the free or λ^2-bound variables. Let the λ-term M have free variables $w_1, \ldots, w_n$ and let $act(M) = x_1 \ldots x_m$. It will be the case that

$$\theta\text{-nf}((M)^\lambda) \equiv (\lambda^2 x_1.\ldots.\lambda^2 x_m.N)$$

for some N which is not an abstraction and which contains no λ^2-bindings. Suppose we want to fix the type of x_1 as $\forall\alpha.(\alpha \to \alpha \to \alpha)$ but we wish the type inference algorithm to find most general open types for the rest of the free and λ^2-bound variables. To accomplish this, we can run the type inference algorithm on the λ-term

$$(\lambda^2 x_1.\mathbf{I}(\lambda^3 w_1.\ldots.\lambda^3 w_n.\lambda^3 x_2.\ldots.\lambda^3 x_m.N))$$

using the type assignment $A = \{x_1 : \forall\alpha.(\alpha\to\alpha\to\alpha)\}$, which will produce a type

$$A(x_1) \to \rho_1 \to \ldots \to \rho_n \to \psi_2 \to \ldots \to \psi_m \to \varphi$$

From this, we can conclude that any type-substitution instance (where ∀-bound variables are unchanged of course) of the following sequent is derivable:

$$A \cup \{w_1 : \rho_1, \ldots, w_n : \rho_n\} \vdash M : A(x_1) \to \psi_2 \to \ldots \to \psi_m \to \varphi$$

At times, we may want to assign more complex closed types to the free and λ^2-bound variables of a λ-term. It would be nice if the type inference algorithm would provide enough information so that we could know if a particular combination of closed types would work. Unfortunately, we do not currently have a method of knowing which closed types can be used without actually trying the type inference algorithm with that set of types assigned to the free and λ^2-bound variables.

References

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

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

[Gir72] J.-Y. Girard. *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieur.* Thèse 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. 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.

[HW88] P. Hudak and P. L. Wadler. Report on the functional programming language Haskell. Technical Report YALEU/DCS/RR656, Yale University, 1988.

[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, Austin, TX, U.S.A., Jan. 24–26, 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, Sophia-Antipolis, France, June 1984. Springer-Verlag.

[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 structure. In *Symposium on Programming*, vol. 19 of *LNCS*, pp. 408–425, Paris, France, 1974. Springer-Verlag.

[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*. Springer-Verlag, Mar. 1993.

[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 λ-calculus are equivalent and undecidable. Tech. Rep. 93-011, Comput. Sci. Dept., Boston Univ., 1993.