Type Inference with Polymorphic Recursion*

Fritz Henglein†
DIKU
University of Copenhagen
Universitetsparken 1
2100 Copenhagen East
Denmark
Internet: henglein@diku.dk

June 28, 1990; revised December 10, 1991

Abstract

The Damas-Milner Calculus is the typed λ-calculus underlying the type system for ML and several other strongly typed polymorphic functional languages such as Miranda[1] and Haskell. Mycroft has extended its problematic monomorphic typing rule for recursive definitions with a polymorphic typing rule. He proved the resulting type system, which we call the Milner-Mycroft Calculus, sound with respect to Milner's semantics, and showed that it preserves the principal typing property of the Damas-Milner Calculus. The extension is of practical significance in typed logic programming languages and, more generally, in any language with (mutually) recursive definitions.

In this paper we show that the type inference problem for the Milner-Mycroft Calculus is *log*-space equivalent to semi-unification, the problem of solving subsumption inequations between first-order terms. This result has been proved independently by Kfoury, Tiuryn, and Urzyczyn. In connection with the recently established undecidability of semi-unification this implies that typability in the Milner-Mycroft Calculus is undecidable.

We present some reasons why type inference with polymorphic recursion appears to be practical despite its undecidability. This also sheds some light on the observed practicality of ML in the face of recent theoretical intractability results.

Finally, we exhibit a semi-unification algorithm upon which a flexible, practical, and implementable type inference algorithm for both Damas-Milner and Milner-Mycroft typing can be based.

1 Introduction

1.1 Polymorphic Recursion

Recently designed languages such as (Standard) ML [MTH90] try to combine the safety of compile-time type checking with the flexibility of declaration-less programming by inferring type information from the program rather than insisting on extensive declarations. ML's type

*To appear in ACM Transactions on Programming Languages and Systems (TOPLAS), 1992.

†This research was performed at New York University (Courant Institute of the Mathematical Sciences) with support by the ONR under contract numbers N00014-85-K-0413 and N00014-87-K-0461.

[1] Miranda^{TM} is a trademark of Research Software Limited.

system, which we call the Damas-Milner Calculus, allows for definition and use of *(parametric) polymorphic* functions; that is, functions that operate uniformly on arguments that may range over a variety of types.

A peculiarity in the Damas-Milner Calculus is that occurrences of a recursively defined function *inside* the body of its definition can only be used *monomorphically* (all of its occurrences have to have identically typed arguments and their results are typed identically), whereas occurrences *outside* its body can be used *polymorphically* (with arguments of different types). For this reason Mycroft [Myc84] and independently Meertens [Mee83] have suggested a *polymorphic* typing rule for recursive definitions that allows for polymorphic occurrences of the defined function in its body. Mycroft has shown that the resulting type system, termed the Milner-Mycroft Calculus here, is sound with respect to Milner's semantics [Mil78] and that the principal typing property of the Damas-Milner Calculus is preserved. The standard unification-based type inference algorithm is not complete for polymorphic recursion, though.

Although the motivation for studying Mycroft's extension to ML's typing discipline may seem rather esoteric and of purely theoretical interest, it stems from practical considerations. In ML many typing problems attributable to the monomorphic recursive definition constraint can be avoided by appropriately nesting function definitions inside the scopes of previous definitions. Since ML provides polymorphically typed **let**-definitions — giving rise to the term **let**-polymorphism — nesting definitions is, indeed, a workable scheme in many cases. Some languages, however, do not provide nested scoping, but only top-level function (or procedure) definitions: e.g., ABC [GMP90], SETL [SDDS86], and Prolog [SS86]. Consequently, all such top-level definitions combined have to be considered a single, generally mutually recursive definition. Adopting ML's monomorphic typing rule for recursive definitions in these languages precludes polymorphic usage of any defined function inside any definition. In particular, since logic programs, as observed in [MO84], can be viewed as massive mutually recursive definitions, using an ML-style type system would eliminate polymorphism from strongly typed logic programming languages almost completely. Mycroft's extension, on the other hand, permits polymorphic usage in such a language setting.

In many cases it is possible to investigate the dependency graph ("call graph") of mutually recursive definitions, process its maximal strong components in topological order, and treat them implicitly as polymorphically typed, nested **let**-definitions. But this is undesirable for several reasons:

1. The resulting typing discipline cannot be explained in a syntax-directed fashion, but is rather reminiscent of data-flow oriented reasoning. This runs contrary to structured programming and program understanding. In particular, finding the source(s) of typing errors in the program text would be made even more difficult than the already problematic attribution of type errors to source code in ML-like languages [JW86,Wan86].

2. The topological processing does not completely capture the polymorphic typing rule. Mycroft reports a mutually recursive definition he encountered in a "real life" programming project that could not be typed in ML, but could be typed by using the polymorphic typing rule for recursive definitions [Myc84, section 8]. Other, similar cases have been reported since.[2]

[2] E.g., on the ML mailing list.

1.2 Example

Consider the following joint definition of functions *map* and *squarelist* in Standard ML, taken from [Myc84].

$$\begin{array}{l}\textbf{fun}\ \mathit{map}\ f\ l = \textbf{if}\ \mathit{null}\ l\ \textbf{then}\ \mathit{nil}\ \textbf{else}\ f\ (\mathit{hd}\ l) :: \mathit{map}\ f\ (\mathit{tl}\ l)\ \textbf{and} \\ \quad \mathit{squarelist}\ l = \mathit{map}\ (\textbf{fn}\ x{:}\ \textbf{int} => x * x)\ l;\end{array}$$

As written this is a simultaneous definition of *map* and *squarelist* even though *squarelist* is not used in the definition of map. An ML type checker produces the types

$$\begin{array}{l}\mathit{map}{:}\ (\textbf{int} \to \textbf{int}) \to \textbf{int list} \to \textbf{int list} \\ \mathit{squarelist}{:}\ \textbf{int list} \to \textbf{int list}\end{array}$$

due to ML's monomorphic recursion rule even though we would expect, as sequential recursive definitions first of *map* and then of *squarelist* would yield, the type of *map* to be

$$\mathit{map}{:}\ \forall\alpha.\forall\beta.(\alpha \to \beta) \times \alpha\ \textbf{list} \to \beta\ \textbf{list}.$$

If we use *map* with an argument type different from **int list** we even get a type error.

1.3 Outline of Paper

In Section 2 we review the formal definition of the Milner-Mycroft Calculus along with the Damas-Milner Calculus, the typing system of the functional core of ML. Section 3 contains an introduction to semi-unification, the problem of solving (equations and) subsumption inequations between first-order terms.

In Section 4, which constitutes the technical core of this paper, type inference and semi-unification are formally connected: we show that typability in the Milner-Mycroft Calculus and semi-unification are *log*-space equivalent. The reduction of Milner-Mycroft typability to semi-unification is presented in several steps. First, we formulate two syntax-directed type inference systems that are equivalent to the canonical type inference system for the Milner-Mycroft Calculus. Using the first-order version of the two we show how to extract from an (possibly untypable) input program a system of equations and inequations between unquantified type expressions that characterizes typability (and more generally the typings) of the program. The converse reduction of semi-unification to Milner-Mycroft typability is also factored into steps. First we introduce pairs into the Milner-Mycroft Calculus and demonstrate how term equations can be encoded as typing constraints in the monomorphic fragment of the Milner-Mycroft Calculus. Then we show how a finite set of term inequations can be represented by the typing constraints of a single recursive definition in the Milner-Mycroft Calculus.

In Section 5 we present a simple, but flexible rewriting system for semi-unification: an initial system of term equations and inequations is rewritten until it reaches a normal form or runs into an error condition. Even though it is possible that the computation does not terminate we know, due to an extended "occurs check", of no known "natural"cases where this actually occurs. We also sketch a flexible graph-theoretic version of the algorithm that supports structure sharing and other optimizations of practical importance. Such a generic semi-unification algorithm can be used as the basis of a practically efficient generic type checker that works well in batch-oriented and interactive programming environments for ML-like languages with or without polymorphic recursion.

In Section 6 we show that, in some sense, type *inference* is no more difficult than type *checking* of (explicitly typed) programs in the presence of polymorphic recursion, and we prove that type inference in both the Damas-Milner Calculus and the Milner-Mycroft Calculus are complexity-theoretically tractable under the — as we argue — reasonable restriction that type expressions not be super-polynomially bigger than the underlying untyped programs. This we offer as a tentative explanation for the observed practicality of ML type inference. Since the same reasoning applies to Milner-Mycroft typing we expect to find type inference with polymorphic recursion equally practical.

Section 7 concludes with a short summary of the main results of the paper and brief remarks on possible future research.

2 Predicative Polymorphic Type Inference Systems

For the purpose of studying polymorphic type inference in isolation from other concerns, we shall restrict ourselves to a notationally minimal programming language, the *(extended) λ-calculus* [Bar84]. Typing disciplines are then defined by inference systems over typing assertions on λ-expressions.[3] For an exposition of the relevance of type theory to programming language design see Cardelli and Wegner [CW85]. Barendregt [Bar84] and Hindley and Seldin [HS86] are in-depth treatises of the λ-calculus. Mitchell [Mit90] is an up-to-date overview of the semantics of the simply typed λ-calculus.

2.1 The Extended Lambda Calculus

The notational conventions used here are fairly standard (see, e.g., [DM82] or [Myc84]). The programming language is an extended λ-calculus with λ-abstraction, application, and nonrecursive and recursive definitions. Our theory is developed only for the "pure" λ-calculus without constants, because constants can be viewed as variables with a given binding in a global environment.

The set Λ of *λ-expressions (expressions)* is defined by the following abstract syntax.

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.e' \mid e'e'' \\ & \mid & \textbf{let } x = e' \textbf{ in } e'' \\ & \mid & \textbf{fix } x.e' \end{array}$$

where x ranges over a countably infinite set V of *variables*. An expression $\lambda x_1.\lambda x_2.\ldots\lambda x_n.e$ may be written $\lambda\vec{x}.e$ where $\vec{x}$ denotes the sequence $x_1 \ldots x_n$. Generally we use this vector notation to refer to sequences and sometimes also to sets of objects; ϵ stands for the empty sequence.

The operational semantics of λ-expressions is defined by *reduction*, which is the reflexive, transitive, compatible[4] closure, $\xrightarrow{*}$, of the *notion of reduction* (see [Bar84, chapter 3]) $\rightarrow$ defined by

[3] Note that we adopt here the "descriptive" view in the sense that programs are defined independently of types; typings *describe* (properties of) such programs. In the "prescriptive" view programs are defined *using* types and typings; typings *prescribe* what constitutes a (well-typed) program in the first place. These views are sometimes also referred to as "Curry" and "Church", respectively, since they may be ascribed to the formulations Curry and Church used for typing in the λ-calculus. For the purposes of type inference they make no difference.

[4] A relation R is compatible if it is closed under taking contexts; that is, $(e_1, e_2) \in R$ implies $(C[e_1], C[e_2]) \in R$ for any context $C[]$ surrounding e_1, respectively e_2.

$$\begin{array}{lcl} (\lambda\, x.e)e' & \rightarrow & e[e'/x] \\ \textbf{let}\ x = e'\ \textbf{in}\ e & \rightarrow & e[e'/x] \\ \textbf{fix}\, x.e & \rightarrow & e[\textbf{fix}\, x.e/x]. \end{array}$$

Here $e[e'/x]$ denotes the expression resulting from substituting e' for all occurrences of x in e.

In the untyped λ-calculus, **let**- and **fix**-expressions can be encoded by λ-abstractions and applications. Since the encodings are not generally typable under the typing disciplines we shall study both forms are introduced as language primitives.

2.2 Types and Typings

Type expressions are formed according to the productions

$$\begin{array}{lcl} \tau & ::= & \alpha \mid \tau' \rightarrow \tau'' \\ \sigma & ::= & \tau \mid \forall\alpha.\sigma' \end{array}$$

where α ranges over an infinite set TV of *type variables* disjoint from V, and $\forall$ is a type variable binding operator. Type expressions derived from τ above are called *(simple) types* and the larger set of type expressions derived from σ are *types schemes*. $FV(\sigma)$ denotes the free type variables in σ. Following Milner [Mil78] we call the bound type variables in a type scheme also *generic* and the free variables in it *nongeneric*. By convention τ always ranges over types and σ over type schemes. Note that the $\forall$-quantifiers in type schemes can only appear as prefixes of type expressions, which is the critical difference from the (impredicative) Second Order λ-calculus [Gir71,Rey74,Mit88,GLT89].

A *type environment* is a mapping from a finite subset of the variables V to type expressions. For type environment A we define

$$A\{x:\sigma\}(y) = \begin{cases} A(y), & y \neq x \\ \sigma, & y = x; \end{cases}$$

that is, the value of $A\{x:\sigma\}$ at x is σ, and at any other value it is identical to A. We say a type variable α occurs free in A if it occurs free in $A(x)$ for some x in the domain of A. We write $FV(A)$ for the set of free type variables in A.

Typings are the well-formed formulae (judgments) of type systems. A typing consists of three parts: a type environment A, an expression e, and a type expression σ, written as $A \vdash e : \sigma$. It should be read as "In the type environment A, the expression e has type σ". Of course, not all typings are acceptable. Acceptability is defined statically by derivability in a specified inference system.

2.3 The Damas-Milner Calculus

The *Damas-Milner Calculus*, in its logical form as a type inference system, was investigated by Damas and Milner [Mil78,DM82,Dam84] on the basis of earlier work by Curry [CF58,Cur69], Morris [Mor68] and Hindley [Hin69]. It encodes the polymorphism that results from the ability in languages such as ML [MTH90], Miranda [Tur86], and Haskell [HW90] to give a **let**-bound variable x a type scheme that is automatically and implicitly instantiated to (possibly different) types for the applied occurrences of x.

Let A range over type environments, x over variables, t over type variables, e and e' over expressions, τ and τ' over types, and σ and σ' over type schemes.

Name	Axiom/rule
(TAUT)	$A\{x:\sigma\} \vdash x:\sigma$
(INST)	$\dfrac{A \vdash e:\forall t.\sigma}{A \vdash e:\sigma[\tau/t]}$
(GEN)	$\dfrac{A \vdash e:\sigma \quad (t \text{ not free in } A)}{A \vdash e:\forall t.\sigma}$
(APPL)	$\dfrac{A \vdash e:\tau' \to \tau \quad A \vdash e':\tau'}{A \vdash (ee'):\tau}$
(ABS)	$\dfrac{A\{x:\tau'\} \vdash e:\tau}{A \vdash \lambda x.e:\tau' \to \tau}$
(LET)	$\dfrac{A \vdash e:\sigma \quad A\{x:\sigma\} \vdash e':\sigma'}{A \vdash \textbf{let } x = e \textbf{ in } e':\sigma'}$
(FIX-M)	$\dfrac{A\{x:\tau\} \vdash e:\tau}{A \vdash \textbf{fix } x.e:\tau}$

Figure 1: The Damas-Milner type inference system DM

$$\text{(FIX-P)}\quad \frac{A\{x:\sigma\} \vdash e:\sigma}{A \vdash \mathbf{fix}\, x.e:\sigma}$$

Figure 2: The polymorphic recursion typing rule of the Milner-Mycroft type inference system MM

The canonical type inference system DM for the Damas-Milner Calculus [DM82] is given in Figure 1. Note that in the last rule, (FIX-M), the type expression associated with the recursively defined x is a *type*, not a *type scheme*. This implies that *all* occurrences of x in e must have one and the same *type* τ.

2.4 The Milner-Mycroft Calculus

The Milner-Mycroft Calculus, presented and investigated by Mycroft [Myc84] and later also by Leiß [Lei87,Lei89b], Kfoury, Tiuryn, and Urzyczyn [KTU88,KTU89], and the author [Hen88, Hen89], differs from the Damas-Milner Calculus only in a more general rule for recursive definitions. It models languages such as Hope [BMS80], Miranda and ABC [GMP90] that permit recursively defined functions to have parameterized *type schemes* that can be instantiated to arbitrary *types* inside the scope of their definition. Hope and Miranda will admit such polymorphically typed recursive definitions only at the top-level, and they require explicit type declarations for such functions. ABC permits no nesting of scopes in the first place, but does not require explicit type declarations. The Milner-Mycroft Calculus does not have either of these restrictions: it requires no explicit declarations and it permits nested polymorphically typed recursive definitions.

The canonical type inference system MM for the Milner-Mycroft Calculus [Myc84] is almost identical to DM, the above presentation of the Damas-Milner Calculus. Instead of the rule (FIX-M) it supplies a *polymorphic recursion* rule, (FIX-P), given in Figure 2, which permits the recursively defined x to be polymorphic.

2.5 Fundamental Properties

Because the type variables in type schemes only range over types, not type schemes, our type systems are called *predicative* [Mit90]. A λ-expression e is *typable (well-typed)* in the Damas-Milner Calculus or in the Milner-Mycroft Calculus, if there is a typing $A \vdash e:\sigma$ derivable in DM or in MM, respectively.

The Milner-Mycroft Calculus preserves many of the desirable properties of the Damas-Milner Calculus. In particular, MM has the *principal typing property* [Myc84], which states that every typable λ-expression has a unique most general type (c.f. [CF58,Hin69,DM82]). Furthermore, it has the *subject reduction property* (due to Curry *et al.* [CF58,CHS72] in its original form for combinatory logic), which states that typings are preserved under the reduction $\rightarrow$ presented in Section 2.1. Finally, it is *sound* with respect to Milner's semantic characterization of well-typing [Mil78,Myc84]; that is, no λ-expression e generates a type error at run-time if e is typable.

Damas-Milner typability is decidable; specifically, typability for the *Curry-Hindley Calculus*, the **let**-free fragment of the Damas-Milner Calculus, is (deterministic) polynomial time

complete (folk theorem) and typability for the full Damas-Milner Calculus is (deterministic) exponential time complete [KM89,Mai90,KTU90a,KMM91]. Yet, Milner-Mycroft typability and semi-unification are *log*-space equivalent (see Section 4), and semi-unification has been shown recursively undecidable [KTU90b].[5]

3 Semi-Unification

Semi-unification is a generalization of both unification and matching with applications in proof theory [Pud88], term rewriting systems [Pur87,KMNS91], polymorphic type inference [Hen88, KTU89,Lei89b], and natural language processing [DR90]. Because of its fundamental nature it can be expected to find even more applications. In this section we review basic definitions and properties of semi-unification. A more detailed account can be found in [Hen89, chapters 3 and 5].

3.1 Basic Definitions

A *ranked alphabet* $\mathcal{A} = (F, a)$ is a finite set F of *function symbols* together with an *arity* function a that maps every element in F to a natural number (possibly zero). A function symbol with arity 0 is also called a *constant*. $\mathcal{A}$ is *monadic* if all its function symbols have arity at most 1, *polyadic* otherwise. The set of *variables* V is a denumerable infinite set disjoint from F. The *terms* over A and V constitute the set $T(\mathcal{A}, V)$ consisting of all strings generated by the grammar

$$M ::= x \mid c \mid f^{(k)}(M^{(1)}, \ldots, M^{(k)})$$

where f is a function symbol from $\mathcal{A}$ with arity $k > 0$ (as indicated by its superscript), c is a constant, and x is any variable from V. The set of variables occurring in M are denoted by $FV(M)$. Two terms M and N are equal, written as $M = N$, if and only if they are identical as strings; e. g., $f(x, y) = f(x, y)$, but $f(x, y) \neq f(c, c)$.

The distinction between monadic and polyadic alphabets is crucial since terms over a monadic alphabet can have at most one variable whereas terms over a polyadic alphabet can contain any number of variables.

A *substitution* S is a mapping from V to $T(\mathcal{A}, V)$ that is the identity on all but a finite subset of V. The set of variables on which S is *not* the identity is the *domain* of S. Every substitution $S : V \to T(\mathcal{A}, V)$ can be naturally extended to $S : T(\mathcal{A}, V) \to T(\mathcal{A}, V)$ by defining

$$S(f(M_1, \ldots, M_k)) = f(S(M_1), \ldots, S(M_k)).$$

A substitution specifies the simultaneous replacement of some set of variables by specific terms. We will write a substitution as a finite mapping on its domain, with the understanding that it acts as the identity on all variables outside its domain. For example, for $S_0 = \{x \mapsto u, y \mapsto v\}$ we have $S_0(f(x, z)) = f(u, z)$. We will write $S|_W$ for the restricted substitution defined by

$$S|_W(x) = \begin{cases} S(x), & x \in W \\ x, & x \notin W \end{cases}$$

[5] The paper [KTU90b] refutes an earlier claim [KTU88] of decidability of Milner-Mycroft typability by the same authors.

If $\vec{M}$ is a sequence of terms and $\vec{x}$ a sequence of variables of equal length then $\vec{M}/\vec{x}$ denotes the substitution that maps every variable in $\vec{x}$ to the term in the corresponding position in $\vec{M}$ (and all other variables to themselves). In this case we write $N[\vec{M}/\vec{x}]$ for the result of its application to term N.

A term M *subsumes* N (or N *matches* M), written $M \leq N$, if there is a substitution R such that $R(M) = N$; e. g., $f(x, y)$ subsumes $f(g(y), z)$ since for $R = \{x \mapsto g(y), y \mapsto z\}$ the equality $R(f(x, y)) = f(g(y), z)$ holds.[6] If N matches M then there is exactly one such R whose domain is contained in the set of variables occurring in M. We call it the *quotient* substitution of M and N and denote it by N/M.

3.2 Systems of Equations and Inequations and their Semi-Unifiers

Given a set of inequations $\mathcal{I} = \{M_1 \stackrel{?}{\leq} N_1, \ldots, M_k \stackrel{?}{\leq} N_k\}$, with $k \geq 0$, a substitution S is a *semi-unifier* of $\mathcal{I}$ if the inequalities $S(M_1) \leq S(N_1), \ldots, S(M_k) \leq S(N_k)$ hold; i.e., there exist substitutions $R_1, \ldots, R_k$ such that the equalities $R_1(S(M_1)) = S(N_1), \ldots, R_k(S(M_k)) = S(N_k)$ hold. $\mathcal{I}$ is *semi-unifiable* if it has a semi-unifier. We shall call $\mathcal{I}$ a *system of inequations (SI)*.

We will also work with *systems of equations and inequations (SEI)* of the form $\{M_{01} \stackrel{?}{=} N_{01}, \ldots, M_{0l} \stackrel{?}{=} N_{0l}, M_1 \stackrel{?}{\leq} N_1 \ldots, M_k \stackrel{?}{\leq} N_k\}$, with $k, l \geq 0$. A substitution S is a semi-unifier of this SEI if $S(M_{01}) = S(N_{01}), \ldots, S(M_{0l}) = S(N_{0l}), S(M_1) \leq S(N_1), \ldots, S(M_k) \leq S(N_k)$ hold. Note that for systems with only equations the notions of semi-unifiability and unifiability coincide. As a notational convenience we may drop the set former brackets in SEI's.

Finally, for convenience we shall also work with tupled inequations in SEI's. We may write $(M_1, \ldots, M_k) \stackrel{?}{\leq} (N_1, \ldots, N_k)$ for $C[M_1, \ldots, M_k] \stackrel{?}{\leq} C[N_1, \ldots, N_k]$ where C is an arbitrary k-ary context.[7] It has a semi-unifier S if and only if there is a (single) substitution R such that $R(S(M_1)) = S(N_1), \ldots, R(S(M_k)) = S(N_k)$.

It is well-known that every (finite) set of equations can be represented as a single equation by tupling. This is *not*, however, the case for inequations. The following proposition makes the connection between semi-unification and unification precise; in particular, it shows that equations can be encoded as inequations.

Proposition 1 *A substitution S is a unifier of the equation $M \stackrel{?}{=} N$ if and only if it is a semi-unifier of $(M, M) \stackrel{?}{\leq} (M, N)$.*

Proof:

only if: If $S(M) = S(N)$ then with the identity substitution Id we have $Id(S(M)) = S(M)$ and $Id(S(M)) = S(N)$ and thus $(S(M), S(M)) \leq (S(M), S(N))$.

if: Conversely, if $(S(M), S(M)) \leq (S(M), S(N))$ then $(R(S(M)) = S(M)$ and $R(S(M)) = S(N)$ for some substitution R. But then $S(M) = S(N)$ follows immediately. ■

In a similar way it can be shown that $M \stackrel{?}{=} N$ is unifiable if and only if $\{M \stackrel{?}{\leq} N, N \stackrel{?}{\leq} M\}$ (or $(M, N) \stackrel{?}{\leq} (N, M)$) is semi-unifiable. But a semi-unifier of the latter is *not* necessarily a unifier of the former.

[6] This definition follows [Hue80] and [Ede85], but is dual to the definition in [LMM87].

[7] A k-ary context is a term with k "holes" in place of subterms. Note that for every $k \geq 0$ there exists a k-ary context as long as the underlying alphabet is polyadic.

It is well-known that every solvable system of *equations* (only) has a most general *unifier* that is unique modulo some equivalence relation on substitutions (c.f. [Ede85] and [LMM87]).[8] A similar result can be obtained for systems of equations and inequations. We phrase this result in algebraic terms. For any subset Φ of V, the preordering $\leq_\Phi$ on the set of substitutions is defined by $S_1 \leq_\Phi S_2 \Leftrightarrow (\exists R)(\forall x \in \Phi) R(S_1(x)) = S_2(x)$. The preordering $\leq_\Phi$ induces an equivalence relation $\cong_\Phi$ on the substitutions by defining $S_1 \cong_\Phi S_2 \Leftrightarrow S_1 \leq_\Phi S_2$ and $S_2 \leq_\Phi S_1$ and a partial ordering on the $\cong_\Phi$-equivalence classes compatible with $\leq_\Phi$.

Theorem 1 *Let $\mathcal{I}$ be a system of inequations over a polyadic alphabet $\mathcal{A}$ whose set of solutions is denoted by $\mathcal{U}$. Let Φ be a subset of V that contains all the variables occurring in $\mathcal{I}$.*

The equivalence classes induced by $\cong_\Phi$ on $\mathcal{U}$, together with an adjoined maximum element Ω, form a complete lattice if and only if $V - \Phi$ is infinite.

A full proof of this theorem is beyond the scope of this paper. It can be found in [Hen89].

Corollary 2 *For every semi-unifiable system of inequations $\mathcal{I}$ there is a substitution S* (most general semi-unifier) *such that*

1. *S is a semi-unifier of $\mathcal{I}$;*
2. *for every semi-unifier S' of $\mathcal{I}$ there is a substitution R such that $R(S(x)) = S'(x)$ for all variables x occurring in $\mathcal{I}$.*

The second property in this corollary *cannot* be replaced by $R \circ S = S'$ and, in contrast to unification, *not* every substitution of the form $R \circ S$ (with S being the most general semi-unifier) is a semi-unifier of $\mathcal{I}$. Theorem 1 yields, together with the reduction of typability to semi-unification in Section 4 an alternative proof of the principal typing property of the Milner-Mycroft Calculus or any other type system that can be reduced to semi-unification in a similar fashion, such as the Damas-Milner Calculus [Hen89].

Semi-unification refers both to the general problem and process of solving SEI's (over polyadic alphabets) for their most general semi-unifiers and to the specific problem of deciding whether semi-unifiers exist (semi-unifiability); we rely on the context for disambiguation.

Whereas semi-unification was long believed to be decidable, Kfoury, Tiuryn and Urzyczyn have recently given an elegant reduction of the *boundedness* problem for deterministic Turing Machines to semi-unification [KTU90b]. By adapting a proof for a similar problem attributed to Hooper [Hoo65] they show that boundedness is undecidable, which implies the undecidability of semi-unification.

Several special cases of semi-unification have been shown to be decidable: uniform semi-unification (solving a *single* term inequation) [Hen88,Pud88,KMNS91], semi-unification over two variables [Lei89a], left-linear semi-unification [KTU89,Hen90] and quasi-monadic semi-unification [LH91]. Since any finite number of term inequations can be effectively reduced to two inequations [Pud88], semi-unification restricted to two inequations remains undecidable, however.

4 Equivalence of Milner-Mycroft Typing and Semi-Unification

In this section we show that Milner-Mycroft typability and semi-unifiability are *log*-space equivalent. Notably, type quantification in the Milner-Mycroft Calculus can be completely characterized by semi-unification, a quantifier-free concept.

[8] Note, however, that there are several different notions of equivalence used in the literature; see [LMM87] for an excellent survey on unification.

In particular, we show that semi-unification can be reduced to Milner-Mycroft typability of expressions that contain no **let**-operator and at most one occurrence of the polymorphic recursion operator **fix**. This implies that the difficulty of type inference is completely subsumed in a single polymorphically typed recursive definition. Neither (polymorphic) **let**-bindings nor nested **let**- and **fix**-bindings add anything to this problem. This contradicts Meertens' expectations that nested declarations and higher-order functions make type inference harder than in ABC [Mee83] and Mycroft's suggestion to prohibit nested polymorphic recursive definitions "due to the exponential cost of analysing nested **fix** definitions" [Myc84]. In connection with the undecidability of semi-unification this implies that Milner-Mycroft typing is undecidable even when restricted to a single recursive definition. Since semi-unification can also be reduced to type inference for ABC [Hen89], type checking ABC programs is also undecidable.

A practical benefit of the reduction of Milner-Mycroft typability to semi-unification is that it shows how a generic semi-unification algorithm can be used as the basis of a flexible type inference algorithm for polymorphically typed languages, with or without polymorphic recursion. This is detailed in Section 5.

Also, we obtain as a by-product explicit *log*-space reductions between unification and typability in the Curry-Hindley Calculus, the **let**-free fragment of the Damas-Milner Calculus. This yields a proof of the apparently long-known "folk theorem" that Curry-Hindley typing is P-complete under *log*-space reductions.

Damas-Milner typability has been characterized by *polymorphic* unification [KM89,KMM91] and by *acyclic* semi-unification [KTU90a]. A characterization of Milner-Mycroft typability by semi-unification has independently been given by Kfoury *et al.* [KTU89]; in fact, Kfoury and Tiuryn have extended it to include the Second Order λ-calculus limited to "rank 2"-derivations [KT90]. Characterizations of type inference by inequality constraints involving quantified types in the Second Order λ-calculus have been given in [Mit88,GRDR88].

We first present the reduction of Milner-Mycroft typability to semi-unification (Section 4.1) and then the converse reduction (Section 4.2).

4.1 Reduction of Typability to Semi-Unification

The reduction from Milner-Mycroft typability to semi-unification was originally reported in [Hen88]. It is broken down as follows. First we present a syntax-directed version for the Milner-Mycroft Calculus that uses type schemes only in type environments (Section 4.1.1). We then show how these type schemes can be encoded in another equivalent type inference system that uses only simple types (Section 4.1.2). Typability in the latter type inference system is finally characterized by systems of equations and inequations between types (Section 4.1.3).

4.1.1 A Syntax-Directed Presentation of the Milner-Mycroft Calculus

The type inference system of Figure 2 is not syntax-directed. This means that the structure of a typing derivation for a given expression e does not directly correspond to the syntactic structure of e itself. This is solely due to the rules (INST) and (GEN) (see Figure 1 in Section 2) since proof steps involving any one of these rules do not "change" the expression in a typing. In a syntax-directed system a derivation for expression e has essentially the same tree structure as a syntax tree for e. The advantage of a syntax-directed inference system is that we can think of a derivation for e as an attribution of its syntax tree.

A *syntax-directed* presentation of the Milner-Mycroft Calculus, called MM', is given in Figure 3. We write $A \vdash_{MM'} e : \tau$ if $A \vdash e : \tau$ is derivable in it. Note that it contains neither (INST)

Let A range over type environments; x over variables; e, e' over λ-expressions; $\vec{\alpha}$ over sequences of type variables; τ, τ' over types, and $\vec{\tau}$ over sequences of types; $\tau[\vec{\tau}/\vec{\alpha}]$ denotes the type τ in which every occurrence of a type variable in $\vec{\alpha}$ is replaced by the corresponding type in $\vec{\tau}$.
The following are the type inference axiom and rule schemes for the syntax-directed Milner-Mycroft type inference system.

Name	Axiom/rule
(TAUT')	$A\{x : \forall\vec{\alpha}.\tau\} \vdash x : \tau[\vec{\tau}/\vec{\alpha}]$
(ABS)	$\dfrac{A\{x : \tau'\} \vdash e : \tau}{A \vdash \lambda x.e : \tau' \to \tau}$
(APPL)	$\dfrac{A \vdash e : \tau' \to \tau \quad A \vdash e' : \tau'}{A \vdash (ee') : \tau}$
(LET')	$\dfrac{A \vdash e : \tau \quad A\{x : \forall\vec{\alpha}.\tau\} \vdash e' : \tau' \quad (\vec{\alpha} = FV(\tau) - FV(A))}{A \vdash \textbf{let}\ x = e\ \textbf{in}\ e' : \tau'}$
(FIX-P')	$\dfrac{A\{x : \forall\vec{\alpha}.\tau\} \vdash e : \tau \quad (\vec{\alpha} = FV(\tau) - FV(A))}{A \vdash \textbf{fix}\ x.e : \tau[\vec{\tau}/\vec{\alpha}]}$

Figure 3: The syntax-directed Milner-Mycroft type inference system MM'

nor (GEN). Instantiation of type schemes to types is restricted to variable occurrences and is incorporated in the new axiom (TAUT'). Generalization of types to type schemes is restricted to **let**- and **fix**-expressions and is incorporated into the new typing rules (LET') and (FIX-P'). Typings are exclusively of the form $A \vdash e : \tau$ where τ is a type, not a type scheme. This is a step in the direction of eliminating constraints involving quantified types, but note that type schemes can still occur in type environments.

We shall now prove that MM' is neither weaker nor stronger than the original system. First we will need a technical proposition stating that quantified type variables can be renamed.

Proposition 3 *For any type environment A, λ-expression e, type expressions σ, σ', and type variables $\vec{\alpha} = \alpha_1 \ldots \alpha_k, \vec{\beta} = \beta_1 \ldots \beta_k$ we have*

1. $A \vdash e : \forall\vec{\alpha}.\sigma \Leftrightarrow A \vdash e : \forall\vec{\beta}.\sigma[\vec{\beta}/\vec{\alpha}]$
2. $A\{x : \forall\vec{\alpha}.\sigma\} \vdash e : \sigma' \Leftrightarrow A\{x : \forall\vec{\beta}.\sigma[\vec{\beta}/\vec{\alpha}]\} \vdash e : \sigma'$

Justified by this proposition we shall assume henceforth that bound type variables occurring in a type scheme σ do not occur anywhere outside of σ.

Theorem 2 *For any type environment A, λ-expression e, type variables $\vec{\alpha} = \alpha_1 \ldots \alpha_k$ not free in A, and type τ we have*

$$A \vdash_{MM} e : \forall\vec{\alpha}.\tau \Leftrightarrow A \vdash_{MM'} e : \tau$$

Corollary 4 *For any $e \in \Lambda$, e is typable in MM if and only if it is typable in MM'.*

This theorem is the extension of Theorem 2.1 in [CDDK86] (which is for the Damas-Milner type inference system DM) to the Milner-Mycroft Calculus; it is essentially identical to Proposition 2.1 in [KTU88]. The theorem is an immediate consequence of the following technical strengthening.

Lemma 5 *For any type environment A, λ-expression e, type variables $\vec{\alpha} = \alpha_1 \ldots \alpha_k$ not free in A, and type τ we have*

$$A \vdash_{MM} e : \forall\vec{\alpha}.\tau \Leftrightarrow (\forall\vec{\tau})\, A \vdash_{MM'} e : \tau[\vec{\tau}/\vec{\alpha}]$$

Proof:

$\Rightarrow$: We proceed by structural induction on derivations in the Milner-Mycroft Calculus of Section 2. We only present two cases, the others being similar.

(TAUT) If we have a trivial derivation involving only (TAUT),

$$A\{x : \forall\vec{\alpha}.\tau\} \vdash x : \forall\vec{\alpha}.\tau$$

then, by (TAUT') in MM' we have immediately

$$A\{x : \forall\vec{\alpha}.\tau\} \vdash x : \tau[\vec{\tau}/\vec{\alpha}].$$

(FIX-P) Assume that $A \vdash \mathbf{fix}\ x.e : \forall\vec{\alpha}.\tau$ is derivable in the Milner-Mycroft Calculus by the (FIX-P) rule; that is,

$$\frac{A\{x : \forall\vec{\alpha}.\tau\} \vdash e : \forall\vec{\alpha}.\tau}{A \vdash \mathbf{fix}x.e : \forall\vec{\alpha}.\tau}$$

By Proposition 3 we may assume that $\vec{\alpha}$ is not free in A and $A\{x : \forall\vec{\alpha}.\tau\}$. By the induction hypothesis we know that $A\{x : \forall\vec{\alpha}.\tau\} \vdash e : \tau$ is derivable in MM', and consequently we get

$$\frac{\begin{array}{l}A\{x : \forall\vec{\alpha}.\tau\} \vdash e : \tau \\ (\text{since } \vec{\alpha} \text{ is not free in } A)\end{array}}{A \vdash \mathbf{fix}x.e : \tau[\vec{\tau}/\vec{\alpha}]}$$

by rule (FIX-P').

$\Leftarrow$: It is sufficient to show $A \vdash_{MM'} e : \tau \Rightarrow A \vdash_{MM} e : \tau$. We shall prove that every axiom and rule scheme in the syntax-directed Milner-Mycroft Calculus is derivable in the (ordinary) Milner-Mycroft Calculus. Again, we only present two cases.

(TAUT') For $A\{x : \forall\vec{\alpha}.\tau\} \vdash x : \tau[\vec{\tau}/\vec{\alpha}]$ derivable by (TAUT') in MM' we have the following typing derivation in the Milner-Mycroft Calculus:

$$\frac{A\{x : \forall\vec{\alpha}.\tau\} \vdash x : \forall\vec{\alpha}.\tau \text{ (TAUT)}}{A\{x : \forall\vec{\alpha}.\tau\} \vdash x : \tau[\vec{\tau}/\vec{\alpha}] \text{ (INST}^k\text{)}}$$

where (INST^k) denotes a k-fold application of rule (INST).

(FIX-P') If we have a derivation in the syntax-directed Milner-Mycroft Calculus ending with the application of the (FIX-P') rule

$$\frac{\begin{array}{l}A\{x : \forall\vec{\alpha}.\tau\} \vdash e : \tau \\ \quad (\vec{\alpha} \text{ not free in } A)\end{array}}{A \vdash \mathbf{fix}x.e : \tau[\vec{\tau}/\vec{\alpha}]}$$

we can construct a corresponding derivation in the Milner-Mycroft Calculus as follows.

$$\dfrac{\dfrac{\dfrac{\begin{array}{l}A\{x : \forall\vec{\alpha}.\tau\} \vdash e : \tau \\ \qquad\qquad (\vec{\alpha} \text{ not free in } A)\end{array}}{A\{x : \forall\vec{\alpha}.\tau\} \vdash e : \forall\vec{\alpha}.\tau \text{ (GEN}^k\text{)}}}{A \vdash \mathbf{fix}x.e : \forall\vec{\alpha}.\tau \text{ (FIX-P)}}}{A \vdash \mathbf{fix}x.e : \tau[\vec{\tau}/\vec{\alpha}] \text{ (INST}^k\text{)}}\ \blacksquare$$

Similar results about "normalizing" typing derivations can be found in [Mit88] and [Car88].

4.1.2 A First-Order Presentation of the Milner-Mycroft Calculus

In MM' type schemes can still occur in type environments. We shall now modify MM' to another syntax-directed version of the Milner-Mycroft Calculus that uses exclusively types, no type schemes. The main question to be addressed is how to represent the difference between generic and nongeneric type variables in a type environment without using explicit quantification. The difference between generic and nongeneric variables is crucial in the (TAUT') rule: if $A(x) = \forall\vec{\alpha}.\tau$ the type of an occurrence of a variable x is a substitution instance of the type τ where *only* the generic type variables in τ are instantiated. Can we eliminate the explicit quantification in A

— i.e., replace A by A' such that $A'(x) = \tau$ — without losing the crucial distinction between generic and nongeneric type variables? The answer to this question and the key observation is already implicit in Milner's original presentation [Mil78] of what is called the Damas-Milner Calculus here: For a **let**- and **fix**-bound variable x in a type environment $A\{x : \tau\}$ we can treat every type variable α in τ as generic *unless* it also occurs in the type of a λ-bound variable in A (see rules (LET') and (FIX')). If x itself is λ-bound then every type variable in τ is nongeneric (see rule (ABSTR)).

Milner's solution to distinguishing generic and nongeneric type variables without explicit quantification is to use sequences of bindings of the form $\lambda x : \tau$, **let** $x : \tau$ and **fix** $x : \tau$ in place of type environments: type variables occurring in λ-bindings are nongeneric, all others are generic. We shall use a different solution that suits our purposes somewhat better.

We say τ is the *underlying (simple) type* of type scheme σ if $\sigma = \forall\vec{\alpha}.\tau$ for some type variables $\vec{\alpha}$. By extension, A is the *underlying simple type environment* of (general) type assigment A' if for every x either both $A(x)$ and $A'(x)$ are undefined or $A(x)$ is the underlying type of $A'(x)$. The underlying simple type environment A of A' together with a sequence of types $\vec{\tau}$ *represents* A', written

$$A, \vec{\tau} \cong A',$$

if $FV(A) - FV(\vec{\tau}) = FV(A')$. Intuitively, the types in $\vec{\tau}$ will simply be the types of all λ-bound variables that have been encountered when processing an expression. The generic type variables in A can be determined to be exactly those that do *not* occur in $\vec{\tau}$. Representations of type environments can be constructed according to the following easily proved proposition.

Proposition 6 *If* $A, \vec{\tau} \cong A'$ *and* $\vec{\alpha} = FV(\tau) - FV(A')$ *then:*

1. $A\{x : \tau\}, (\vec{\tau}, \tau) \cong A'\{x : \tau\}$;
2. $A\{x : \tau\}, \vec{\tau} \cong A'\{x : \forall\vec{\alpha}.\tau\}$.

By replacing type environments by their underlying simple type assignments together with the types of the λ-bound variables in the type environment it is possible to give a first-order presentation of the Milner-Mycroft Calculus. Furthermore, the restriction of the substitution to generic variables in rule (TAUT') can be formulated as a single subsumption inequality between types. Recall that $A'\{x : \forall\vec{\alpha}.\tau\} \vdash x : \tau'$ is derivable in the syntax-directed presentation of the Milner-Mycroft Calculus if and only if there is a substitution R such that $R(\tau) = \tau'$ and $R(\beta) = \beta$ for every $\beta \in FV(A'\{x : \forall\vec{\alpha}.\tau\})$. For $A\{x : \tau\}, \vec{\tau} \cong A'\{x : \forall\vec{\alpha}.\tau\}$ this is equivalent to $R(\tau) = \tau'$ and $R(\vec{\tau}) = \vec{\tau}$; i.e., $(\tau, \vec{\tau}) \leq (\tau', \vec{\tau})$ using the tupling notation of Section 3. Consequently it is possible to replace (TAUT') by the rule

$$\text{(TAUT")} \qquad A\{x : \tau\}, \vec{\tau} \vdash \tau' \quad \text{if } (\tau, \vec{\tau}) \leq (\tau', \vec{\tau}).$$

The other rules are easily adapted from Figure 3. The resulting type system MM" is displayed in Figure 4.

We write $A, \vec{\tau} \vdash_{MM"} e : \tau$ if $A, \vec{\tau} \vdash e : \tau$ is derivable in MM". Derivable typings in this type system faithfully describe typings in MM', the syntax-directed presentation of the Milner-Mycroft Calculus, as expressed in the following theorem.

Theorem 3 *For all* $A, A', e, \tau, \vec{\tau}$, *if* $A, \vec{\tau} \cong A'$ *then*

$$A' \vdash_{MM'} e : \tau \Leftrightarrow A, \vec{\tau} \vdash_{MM"} e : \tau.$$

Let A range over simple type environments; x over variables; e, e' over λ-expressions; τ, τ' over types, and $\vec{\tau}$ over sequences of types. The following are the type inference axiom and rule schemes for the first-order Milner-Mycroft type inference system.

Name	Axiom/rule
(TAUT")	$\dfrac{(\tau, \vec{\tau}) \leq (\tau', \vec{\tau})}{A\{x:\tau\}, \vec{\tau} \vdash x : \tau'}$
(ABS")	$\dfrac{A\{x:\tau_x\}, (\vec{\tau}, \tau_x) \vdash e : \tau \quad \tau' = \tau_x \to \tau}{A, \vec{\tau} \vdash \lambda x.e : \tau'}$
(APPL")	$\dfrac{A, \vec{\tau} \vdash e : \tau \quad A, \vec{\tau} \vdash e' : \tau' \quad \tau = \tau' \to \tau''}{A, \vec{\tau} \vdash (ee') : \tau''}$
(LET")	$\dfrac{A, \vec{\tau} \vdash e : \tau \quad A\{x:\tau_x\}, \vec{\tau} \vdash e' : \tau' \quad \tau = \tau_x, \tau' = \tau''}{A, \vec{\tau} \vdash \textbf{let}\ x = e\ \textbf{in}\ e' : \tau''}$
(FIX-P")	$\dfrac{A\{x:\tau_x\}, \vec{\tau} \vdash e : \tau \quad \tau_x = \tau, (\tau, \vec{\tau}) \leq (\tau', \vec{\tau})}{A, \vec{\tau} \vdash \textbf{fix}\ x.e : \tau'}$

Figure 4: The first-order presentation MM" of the Milner-Mycroft Calculus

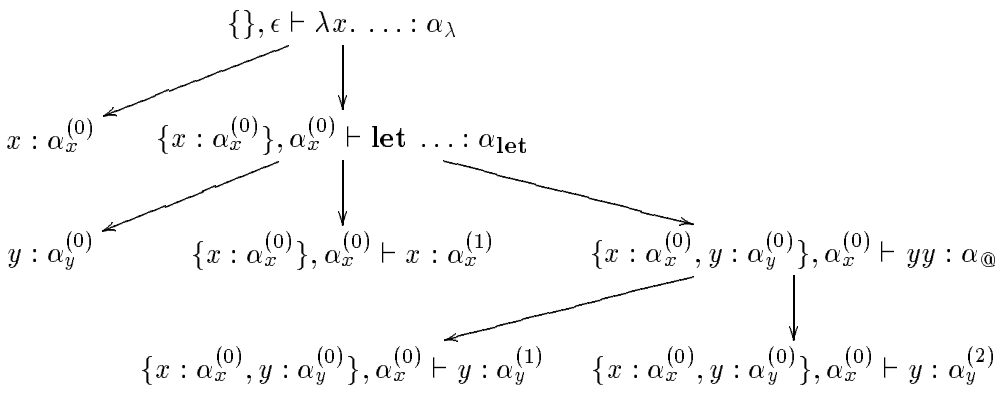

Figure 5: Typing derivation template for $e_0 = \lambda x.\textbf{let}\ y = x\ \textbf{in}\ yy$

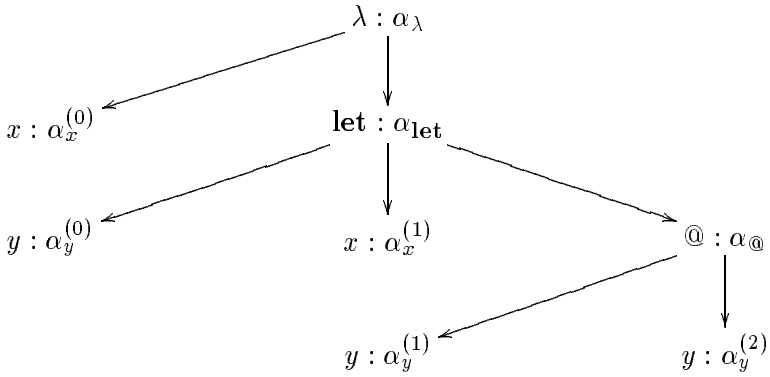

Figure 6: Syntax tree of e_0 attributed with unique type variables

The left-to-right implication can be proved by induction on typing derivations in MM', and the reverse implication by induction on typing derivations in MM". The main technical insight necessary for the proof is the justification for the encoding of rule (TAUT') by rule (TAUT") above, which eliminates the explicit distinction between generic and nongeneric variables.

Corollary 7 *Let A be the underlying simple type environment of A' and $\vec{\alpha} = FV(A')$. Then $A' \vdash_{MM'} e : \tau \Leftrightarrow A, \vec{\alpha} \vdash e : \tau$.*

4.1.3 Characterization by Equations and Inequations

The reduction of Milner-Mycroft typing to semi-unification is almost complete. Let us assume $A, \vec{\tau}$ and e are given. If we associate a unique type variable $\alpha_{e'}$ with every occurrence e' of an expression or variable in e then there is a unique derivation "template" of $A, \vec{\tau} \vdash e : \alpha_e$ *disregarding the side conditions on types* (which are set off to the right in the rules of Figure 4). Consider, for example, the λ-expression $e_0 = \lambda x.\textbf{let}\ y = x\ \textbf{in}\ yy$. The typing derivation template for e_0 is depicted in Figure 5. Such a template is isomorphic to the syntax tree of the underlying expression where every node is attributed with a unique type variable; for e_0 see Figure 6.

A substitution mapping type variables to types maps such a template to a valid typing derivation if and only if it satisfies all the side conditions required by the typing rules. For e_0

$SEI(A, \vec{\alpha}, e) =$

case e **of**

x: $(\{(A(x), \vec{\alpha}) \stackrel{?}{\leq} (\alpha_e, \vec{\alpha})\}, \alpha)$
  where α_e is a fresh type variable;

$\lambda x.e'$: $(\mathcal{I}' \cup \{\alpha_x \to \alpha_{e'} \stackrel{?}{=} \alpha_e\}, \alpha_e)$
  where α_x, α_e are fresh type variables and
    $(\mathcal{I}', \alpha_{e'}) = SEI(A\{x : \alpha_x\}, (\vec{\alpha}, \alpha_x), e')$;

$e'e''$: $(\mathcal{I}' \cup \mathcal{I}'' \cup \{\alpha_{e''} \to \alpha_e \stackrel{?}{=} \alpha_{e'}\}, \alpha_e)$
  where α_e is a fresh type variable,
    $(\mathcal{I}', \alpha_{e'}) = SEI(A, \vec{\alpha}, e')$ and
    $(\mathcal{I}'', \alpha_{e''}) = SEI(A, \vec{\alpha}, e'')$;

let $x = e'$ **in** e'': $(\mathcal{I}' \cup \mathcal{I}'' \cup \{\alpha_x \stackrel{?}{=} \alpha_{e'}, \alpha_{e''} \stackrel{?}{=} \alpha_e\}, \alpha_e)$
  where α_x, α_e are fresh type variables,
    $(\mathcal{I}', \alpha_{e'}) = SEI(A, \vec{\alpha}, e')$ and
    $(\mathcal{I}'', \alpha_{e''}) = SEI(A\{x : \alpha_x\}, \vec{\alpha}, e'')$;

fix $x.e'$: $(\mathcal{I}' \cup \{\alpha_x \stackrel{?}{=} \alpha_{e'}, (\alpha_{e'}, \vec{\alpha}) \stackrel{?}{\leq} (\alpha_e, \vec{\alpha})\}, \alpha_e)$
  where α_x, α_e are fresh type variables and
    $(\mathcal{I}', \alpha_{e'}) = SEI(A\{x : \alpha_x\}, \vec{\tau}, e')$

end case.

Figure 7: Definition of $SEI(A, \vec{\alpha}, e)$

the side conditions are:

$$\begin{array}{rcl}
(\alpha_y^{(0)}, \alpha_x^{(0)}) & \stackrel{?}{\leq} & (\alpha_y^{(1)}, \alpha_x^{(0)}) \\
(\alpha_y^{(0)}, \alpha_x^{(0)}) & \stackrel{?}{\leq} & (\alpha_y^{(2)}, \alpha_x^{(0)}) \\
\alpha_y^{(1)} & \stackrel{?}{=} & \alpha_y^{(2)} \to \alpha_{@} \\
(\alpha_x^{(0)}, \alpha_x^{(0)}) & \stackrel{?}{\leq} & (\alpha_x^{(1)}, \alpha_x^{(0)}) \\
\alpha_y^{(0)} & \stackrel{?}{=} & \alpha_x^{(1)} \\
\alpha_{\mathbf{let}} & \stackrel{?}{=} & \alpha_{@} \\
\alpha_{\lambda} & \stackrel{?}{=} & \alpha_x^{(0)} \to \alpha_{\mathbf{let}}
\end{array}$$

Notice that the side conditions for *any* λ-expression form a *system of equations and inequations* over the alphabet $\mathcal{A}_2 = \{\to^{(2)}\}$, which contains $\to$ as the only function symbol. This follows from the fact that every syntactic construct has exactly one typing rule in Figure 4 that is applicable if and only if the side conditions on types are satisfied since every type meta-variable occurs at most once in its antecedent(s) and conclusion.

Formally, for given simple type environment A, sequence of type variables $\vec{\alpha}$ and λ-expression e we define a system of equations and inequations $\mathcal{I}$ together with a type variable α_e by $(\mathcal{I}, \alpha_e) = SEI(A, \vec{\alpha}, e)$, using the procedure SEI given in Figure 7. For λ-bound variables x we can generate the equational constraint $A(x) = \alpha_e$ instead of $(A(x), \vec{\alpha}) \stackrel{?}{\leq} (\alpha_e, \vec{\alpha})$ in Figure 7 since $\vec{\alpha}$ is guaranteed to contain $A(x)$ and $(A(x), A(x)) \stackrel{?}{\leq} (\alpha_e, A(x))$ is equivalent to $A(x) = \alpha_e$ (see

Proposition 1 in Section 3).

Theorem 4 *Let* $(\mathcal{I}, \alpha_e) = SEI(A, \vec{\alpha}, e)$. *Then* $A, \vec{\alpha} \vdash_{MM''} e : \tau$ *if and only if there is a semi-unifier* S *of* $\mathcal{I}$ *such that* $S(\alpha_e) = \tau$.

The left-to-right implication is proved by induction on typing derivations in MM", and the converse implication is proved by induction on e. The details of the proof are self-evident and are thus omitted.

Corollary 8 *Expression* e *is Milner-Mycroft typable under* $A, \vec{\alpha}$ *if and only if the system of equations and inequations generated by* $SEI(A, \vec{\alpha}, e)$ *is semi-unifiable.*

Damas-Milner typing can analogously be reduced to semi-unification. There are more equational constraints and fewer inequality constraints in comparison to Milner-Mycroft typing. The only change is in the constraint generated from a **fix**-expression: the inequation $(\alpha_{e'}, \vec{\alpha}) \stackrel{?}{\leq} (\alpha_e, \vec{\alpha})$ in Figure 7 is replaced by the equality $\alpha_{e'} = \alpha_e$. For the Curry-Hindley Calculus all extracted constraints are equational and the resulting problem is unification, which can be solved efficiently by any number of unification algorithms [Hue76,PW78,MM82]. Since the additional inequational constraints for the Damas-Milner Calculus seemed rather innocuous at first sight, it was long believed that Damas-Milner typability was equally efficient (e.g., [Lei83,MH88]) before it was refuted by Kanellakis, Mairson and Mitchell [KM89,Mai90,KMM91], and Kfoury, Tiuryn, and Urzyczyn [KTU90a].

This concludes our reduction from typability in the canonical Milner-Mycroft type inference system to semi-unification. It is left to ascertain that the reductions can be implemented in logarithmic (auxiliary) space.

Theorem 5 *Milner-Mycroft typability is* log*-space reducible to semi-unifiability.*

Proof: Given a *syntactically well-formed* λ-expression e (e.g., in fully parenthesized notation) and a type environment A' an equivalent SEI can be constructed by composition of the following *log*-space reductions:

1. Generation of the underlying simple type environment A of A' and the free type variables $\vec{\alpha}$ in A';

2. generation of the system of equations and inequations as returned by $SEI(A, \vec{\alpha}, e)$.

To accomplish the generation of the equations and inequations in logarithmic space we cannot use the procedure in Figure 7 literally since it implicitly uses a stack and passes the list of type variables $\vec{\alpha}$ as an argument. Since the values of A and $\vec{\alpha}$ relevant for any subexpression e' occurring in e can be determined in logarithmic space from the input and the location of e' itself they need not be stored explicitly, however. Furthermore, since the call structure of SEI follows the syntactic structure of e, storing the stack explicitly is not necessary, either.[9] This makes it possible to implement $SEI(A, \vec{\alpha}, e)$ using only logarithmic (auiliary) space.

By corollaries 4, 7 and 8, the generated SEI is semi-unifiable if and only if e is Milner-Mycroft typable. ■

[9] Note that checking for syntactic correctness of e is *not* part of this reduction: e is *assumed* to be syntactically correct and given as a (representation of) a correct abstract syntax tree.

Let A range over type environments; e and e' over Λ^p; and τ, τ' and τ'' over types. The typing rules for pairs and projections are as follows.

$$\text{(PAIR)} \quad \frac{A \vdash e : \tau \qquad A \vdash e' : \tau'}{A \vdash (e, e') : (\tau, \tau')_{\tau''}}$$

$$\text{(FST)} \quad \frac{A \vdash e : (\tau, \tau')_{\tau}}{A \vdash e.1 : \tau}$$

$$\text{(SND)} \quad \frac{A \vdash e : (\tau, \tau')_{\tau'}}{A \vdash e.2 : \tau'}$$

Figure 8: The typing rules for pairs and projections

4.2 Reduction of Semi-Unification to Milner-Mycroft Typability

Our reduction of semi-unification to Milner-Mycroft typability proceeds in several stages. First we introduce pairs and pair types into the Milner-Mycroft Calculus (Section 4.2.1). We then identify first-order terms with certain λ-representations of (Section 4.2.2) and define encodings of term equations (Section 4.2.3). This, coincidentally, produces a *log*-space reduction of unification to Curry-Hindley type inference. Finally we encode whole systems of inequations as **fix**-expressions (Section 4.2.4).

We believe the use of pairing makes for a more conspicuous presentation of this reduction. Somewhat different reductions are presented in [Hen89] and [KTU89].

4.2.1 Milner-Mycroft Calculus with Pairing

It will be convenient to work with λ-expressions with pairs and the corresponding component projections. We use the "standard" encoding of pairs, i.e.

$$\begin{aligned} (e, e') &= \lambda x.xee' \\ e.1 &= e(\lambda x.\lambda y.x) \\ e.2 &= e(\lambda x.\lambda y.y) \end{aligned}$$

where x is not free in e or e'. Correspondingly we define

$$(\tau, \tau')_{\tau''} = (\tau \to \tau' \to \tau'') \to \tau''.$$

With this notation it is easy to check that the rules for pairs and projections in Figure 8 are derivable in MM. In fact *every* typing for pairs and projections is an instance of these rules.

Note that τ'' must be equal to the type of the result component in a projection expression $e.1$ or $e.2$. In particular, if x is a variable of *simple* type $(\tau, \tau')_{\tau''}$ for some τ'', then for *both* $x.1$ and $x.2$ to be type correct it *must* be the case that $\tau = \tau'$. This is a stronger typing condition than one would expect from pairs as language primitives! To illustrate this problem consider the expression $e = \lambda x.(x.1, x.2)$. Disregarding the subscripts in Figure 8 we might expect e to have

type $\forall\alpha.\forall\beta.(\alpha,\beta)\to(\alpha,\beta)$. With $I=\lambda x.x$ and $K=\lambda x.\lambda y.x$ as usual, $e(I,K)$ should then have the same type as (I,K): $\forall\alpha.\forall\beta.\forall\gamma.(\alpha\to\alpha,\beta\to\gamma\to\beta)$. Because of the projections on both components of x expression e has only type $\forall\alpha.(\alpha,\alpha)_\alpha\to(\alpha,\alpha)_\alpha$, however.[10] The expression $e(I,K)$ is not even typable!

This problem motivated Mairson [Mai90] to use another representation of pairs whose typing rules faithfully reflect that the component types of pairs are independent of each other.[11] This is not necessary in our case since we will not encounter the above anomolous situation. Mairson's representation can be used to reduce typing with pairs to typing without pairs. Alternatively, we can keep the standard representation of pairs and check that the steps in the following subsections are also correct for them. So we simply write (τ,τ') instead of $(\tau,\tau')_{\tau''}$ and use pairs, projections, pair types and the rules of Figure 8 *without* the subscripts as language primitives of the Milner-Mycroft Calculus enriched with pairs.

4.2.2 Representation of Terms

It can be shown that semi-unifiability over any (polyadic or monadic) alphabet can be reduced to semi-unifiability over alphabet $\mathcal{A}_2$, which contains only a single, binary function symbol $f^{(2)}$ [Hen89]. Since the name of the function symbol is irrelevant we simply write (M,N) for the term $f(M,N)$ and *identify* terms with λ-expressions built from variables and pairing alone.

We extend a simple type environment A to a mapping from terms to types as follows:

$$A((M_1,M_2))=(A(M_1),A(M_2)).$$

The following lemmas are needed in the encodings of equations and inequations. They imply that even though A has different domain and range we may treat it as a substitution on terms.

Lemma 9 *For all simple type environments A, terms M and types τ*

$$A\vdash_{MM'}M:\tau\Leftrightarrow\tau=A(M)$$

.

Lemma 10 *Let A be a simple type environment; let M and N be terms.*

1. *If $A(M)=A(N)$ then M and N are unifiable.*
2. *If $A(M)\leq A(N)$ then $M\stackrel{?}{\leq}N$ is semi-unifiable.*

4.2.3 Encoding of Equations

Let us write $e\doteq e'$ for the expression $\lambda y.(ye,ye')$ where y does not occur freely in e or e'.

Theorem 6 *Let M and N be terms and $\vec{x}=FV(M)\cup FV(N)$. The equation $M\stackrel{?}{=}N$ is unifiable if and only if $\lambda\vec{x}.M\doteq N$ is Curry-Hindley typable.*

Proof:

[10] This is not the principal type, but it can be treated as principal *relative* to any application of e to a pair.
[11] It does not faithfully encode the operational semantics of pairing and projections, however.

$\Rightarrow$: Let S be a unifier of M and N. And let A be an arbitrary simple type environment whose domain contains all the variables occurring in $S(M)$ or $S(N)$. Define $A'(x) = A(S(x))$ for all x in the domain of A. We have $A' \vdash M : A(S(M))$ and $A' \vdash N : A(S(N))$ by Lemma 9. Since S is a unifier of M and N the types $A(S(M))$ and $A(S(N))$ are equal, and consequently $M \doteq N$ is typable under type environment A'. Since A' is simple, $\lambda\vec{x}.M \doteq N$ is also typable.

$\Leftarrow$: If $\lambda\vec{x}.M \doteq N$ is typable then there is a simple type environment A and a type τ such that $A \vdash M : \tau$ and $A \vdash N : \tau$. By Lemma 9 we have $A(M) = \tau = A(N)$, and thus, by Lemma 10, M and N are unifiable. ■

Corollary 11 *Curry-Hindley typability and unification are* log*-space equivalent; in particular, Curry-Hindley typability is P-complete.*

Proof: This follows from P-completeness of unification [DKM84]. ■

4.2.4 Encoding of Inequations

To gain some intuition about the encoding of systems of inequations let us consider a single inequation $M \stackrel{?}{\leq} N$. Inequations arise in particular in the typing of **fix**-bound variables in type inference system MM" (Figure 4). Note that the types for y and e must be equal in any well-typed expression of the form $\mathbf{fix}\, y.e$. Now if we can "force" e to have the type of term M and if we can "hide" (in the sense that it does not affect the type of e) somewhere in e the λ-encoding $y \doteq N$, then the y in $y \doteq N$ is bound to have the same type as N, but by the typing rules for **fix** the type of the occurrence y must also be a substitution instance of the type of e; i.e. the type of N must be a substitution instance of the type of M.

Since M and N generally contain free variables we have to be a little bit more careful than this. To make sure that different occurrences of a free variable in M have the same type everywhere (which corresponds to a semi-unifier uniformly applying the same substitution to all occurrences of a variable), the variables in M and N have to be λ-bound some place, as was the case for encodings of equations (for the same reason, by the way). The λ-bindings for these variables cannot go outside of the whole expression, as in $\lambda\vec{x}.\mathbf{fix}\, y.e$, since this would mean that the **fix**-binding is in the scope of the λ-bindings, and essentially *no* type variable in the type of e could be instantiated. Consequently the place where the λ-bindings have to go is just after the **fix**-binding: $\mathbf{fix}\, y.\lambda\vec{x}.e$. This in turn complicates the encoding of the equation $y \doteq N$ above, but not by much. The details are below.

Theorem 7 *Semi-unification is* log*-space reducible to Milner-Mycroft typability; specifically, for every system of inequations $\mathcal{I}$ there is a* log*-space computable expression e of the form* $\mathbf{fix}\, y.e'$ *where e' is is* **let**- *and* **fix**-*free such that $\mathcal{I}$ is semi-unifiable if and only if e is Milner-Mycroft typable.*

Proof: Without loss of generality we may assume that $\mathcal{A} = \mathcal{A}_2$. We show how to reduce a system of two inequations $\{M_1 \leq N_1, M_2 \leq N_2\}$. This is sufficient by a result of Pudlak [Pud88]. Alternatively, the proof is easily generalized to any number of (equations and) inequations.

Consider the expression $e =$

$$\mathbf{fix}\, f.\lambda\vec{x}.K(M_1, M_2)(\lambda\vec{y}.((f\,\vec{y}).1 \doteq N_1), \lambda\vec{y}.((f\,\vec{y}).2 \doteq N_2)).$$

where $\vec{x} = FV(M_1) \cup FV(M_2) \cup FV(N_1) \cup FV(N_2)$ and $K = \lambda x.\lambda y.x$. Clearly this expression can be constructed in logarithmic space from the given SEI. By analysis of MM" we find that e is Milner-Mycroft typable if and only if there are types $\vec{\tau}, \vec{\tau}', \vec{\tau}'', \tau_1, \tau_2, \tau_1', \tau_2', \tau_1'', \tau_2''$ such that

$$\begin{array}{rcl}
\{f : \vec{\tau} \to (\tau_1, \tau_2)\}, \epsilon & \vdash & f : \vec{\tau'} \to (\tau_1', \tau_2') \\
\{f : \vec{\tau} \to (\tau_1, \tau_2)\}, \epsilon & \vdash & f : \vec{\tau''} \to (\tau_1'', \tau_2'') \\
\{\vec{x} : \vec{\tau}\}, \vec{\tau} & \vdash & M_1 : \tau_1 \\
\{\vec{x} : \vec{\tau}\}, \vec{\tau} & \vdash & M_2 : \tau_2 \\
\{\vec{x} : \vec{\tau}\}, \vec{\tau} & \vdash & N_1 : \tau_1' \\
\{\vec{x} : \vec{\tau}\}, \vec{\tau} & \vdash & N_2 : \tau_2''
\end{array}$$

are derivable in MM" (ϵ denotes the empty sequence). Let us denote $\{\vec{x} : \vec{\tau}\}$ by A. By Lemma 9 we have

$$\begin{array}{rcl}
\tau_1 & = & A(M_1) \\
\tau_2 & = & A(M_2) \\
\tau_1' & = & A(N_1) \\
\tau_2'' & = & A(N_2).
\end{array}$$

Furthermore, by rule (TAUT") in MM" the typings for f are derivable if and only if

$$\begin{array}{rcl}
(A(\vec{x}), A(M_1), A(M_2)) & \leq & (\vec{\tau'}, A(N_1), \tau_1'') \\
(A(\vec{x}), A(M_1), A(M_2)) & \leq & (\vec{\tau''}, \tau_2', A(N_2))
\end{array}$$

Since $\vec{\tau}', \vec{\tau}'', \tau_1'', \tau_2'$ are uniquely determined by the quotient substitution of the following inequalities, e is typable if and only if there is a type environment A such that

$$\begin{array}{rcl}
A(M_1) & \leq & A(N_1) \\
A(M_2) & \leq & A(N_2).
\end{array}$$

Finally, by Lemma 10 this means that e is Milner-Mycroft typable if and only if $\{M_1 \leq N_1, M_2 \leq N_2\}$ is semi-unifiable. ■

Corollary 12 *The following three problems are* log*-space equivalent:*

1. *Milner-Mycroft typability;*
2. *semi-unifiability;*
3. *Milner-Mycroft typability restricted to expressions containing a single (top-level)* **fix***-operator and no* **let***-expression.*

Proof: The steps (1) ⇒ (2) and (2) ⇒ (3) are proved in Theorems 5 and 7, respectively; (3) ⇒ (1) is trivial. ■

5 Semi-Unification Algorithms

In section 4 we have shown that semi-unification is at the heart of polymorphic type inference in the Milner-Mycroft Calculus. In this section we address the problem of computing most general semi-unifiers. Computing the most general semi-unifier of a system of equations and inequations (SEI's) corresponds directly to computing the principal typing of the expression from which the equations and inequations are generated as in Figure 7.

The basic question that must be asked at this point is why we would be interested in or even consider algorithms for semi-unification, given that semi-unification and thus type inference with polymorphic recursion is undecidable. We defer a discussion of this question to Section 6. Suffice it to say at this point that we expect type inference with polymorphic recursion to be just as practical as ML type inference.

We present two algorithms for computing the most general semi-unifier of an SEI in this chapter. The first one is an SEI-rewriting system whose partial correctness follows from a soundness and completeness theorem that shows that the class of solutions is invariant under rewritings (Section 5.1). The second algorithm is a graph-theoretic version of the SEI-rewriting algorithm. By permitting structure sharing in *arrow graphs*, which are term graphs with some additional structure, we expect this algorithm to yield practically efficient implementations. It has also been helpful in analyzing termination properties and complexity of restricted semi-unification problems [Hen90,LH91] (Section 5.2). A third, functional algorithm can be found in [Hen88]. These three algorithms can be viewed as manifestations (or implementations) of a single abstract algorithm. A discussion of a flexible implementation strategy (Section 5.3) for type checkers of polymorphic languages based on a generic semi-unification algorithm concludes this section.

5.1 SEI-Rewriting Algorithm

In this section we present a basic, implementable rewriting system computing most general semi-unifiers. It is a natural and straightforward extension of the rewriting system for most general unifiers that reportedly goes back to Herbrand [Her68] and which was used by Martelli and Montanari as the starting point for the development of efficient unification algorithms [MM82]. It has a novel "extended occurs check" that catches in practice most of the cases in which the type inference algorithms of Meertens [Mee83, Algorithm AA] and Mycroft [Myc84] or the relaxation algorithm of Chou [Cho86] enter an infinite loop.[12] A similar rewriting system using an occurs check clause similar to ours can be found in the work of Leiß [Lei89b].

The SEI rewriting system is given in Figure 9. It preserves semi-unifiers in a sense that we shall make precise below.

Definition 1 *Let $\Rightarrow$ be a reduction relation on systems of equations and inequations and let $V = FV(\mathcal{I})$ be the set of variables occurring in SEI $\mathcal{I}$.*

1. *The relation $\Rightarrow$ is* sound *if for all $\mathcal{I}, \mathcal{I}'$ such that $\mathcal{I} \Rightarrow \mathcal{I}'$ and for every semi-unifier S' of $\mathcal{I}'$ there is a semi-unifier S of $\mathcal{I}$ such that $S|_V = S'|_V$.*
2. *The relation $\Rightarrow$ is* complete *if for all $\mathcal{I}, \mathcal{I}'$ such that $\mathcal{I} \Rightarrow \mathcal{I}'$ and for every semi-unifier S of $\mathcal{I}$ there is a semi-unifier S' of $\mathcal{I}'$ such that $S|_V = S'|_V$.*

[12] In fact we know of no explicitly constructed semi-unification instance where our rewriting system does *not* terminate. Note that such instances must exist [KTU90b].

Given an SEI S with k inequations we initially tag all the inequality symbols with distinct "colors" $1, \ldots, k$ to indicate to which inequation they belong. This is done by superscripts to the inequality symbol; e.g., $\leq^{(1)}$. Then nondeterministically choose an equation or inequation and take a rewriting action depending on its form.[a]

1. $f(M_1, \ldots, M_k) = f(N_1, \ldots, N_k)$:

   Replace by the equations $M_1 = N_1, \ldots, M_k = M_k$.

2. $f(M_1, \ldots, M_k) = g(N_1, \ldots, N_l)$ where f and g are distinct function symbols:

   Replace current SEI by $\Box$ (function symbol clash).

3. $f(M_1, \ldots, M_k) = x$:

   Replace by $x = f(M_1, \ldots, M_m)$.

4. $x = f(M_1, \ldots, M_k)$ where x occurs in at least one of $M_1, \ldots, M_k$:

   Replace current SEI by $\Box$ (occurs check).

5. $x = M$ where x does not occur in M, but occurs in another equation or inequation:

   Replace x by M in all other equations or inequations.

6. $x = x$:

   Delete it.

7. $f(M_1, \ldots, M_k) \leq^{(i)} f(N_1, \ldots, N_k)$:

   Replace by inequations $M_1 \leq^{(i)} N_1, \ldots, M_k \leq^{(i)} M_k$.

8. $x \leq^{(i)} M$ and $x \leq^{(i)} N$:

   Delete one of the two inequations and add the equation $M = N$.

9. $f(M_1, \ldots, M_k) \leq^{(i_0)} x$ and there are variables $x_0, \ldots, x_n$ such that $x = x_0$, $x_i \leq^{(j_i)} x_{i+1}$ are inequations in the current SEI for $0 \leq i \leq n-1$, and there exists an i such that x_n occurs in M_i:

   Replace current SEI by $\Box$ (extended occurs check).

10. $f(M_1, \ldots, M_k) \leq^{(i_0)} x$ and there is no sequence of variables $x_0, \ldots, x_n$ such that $x = x_0$, $x_i \leq^{(j_i)} x_{i+1}$ are inequations in the current SEI for $0 \leq i \leq n-1$ and x_n occurs in some M_i:

    Add the equation $x = f(x'_1, \ldots, x'_k)$ where $x'_1, \ldots, x'_k$ are "fresh" variables not occurring in the current SEI.

[a] The special symbol $\Box$ denotes an unsolvable SEI.

Figure 9: SEI-rewriting specification

Informally speaking, soundness expresses that a reduction step does not add semi-unifiers, and completeness means that no semi-unifiers are lost in a reduction step.

Proposition 13 *The reduction relation defined by the SEI-rewriting system (in Figure 9) is sound and complete.*

Proof: Induction on the number of rewriting steps. ■

Any SEI $\mathcal{I}$ is in *normal form* with respect to a reduction relation $\Rightarrow$ if there is no $\mathcal{I}'$ such that $\mathcal{I} \Rightarrow \mathcal{I}'$. If an SEI is in normal form with respect to our SEI rewriting system it is easy to extract a most general semi-unifier from it.

Proposition 14 *Let $\mathcal{I}$ be a system of equations and inequations in normal form with respect to the reduction relation defined by the SEI rewriting system in Figure 9.*

If $\mathcal{I} = \{x_1 = M_1, \ldots, x_k = M_k, y_1 \leq N_1, \ldots, y_l \leq N_l\}$ then the substitution $S = \{x_1 \mapsto M_1, \ldots, x_k \mapsto M_k\}$ is a most general idempotent semi-unifier of $\mathcal{I}$.

Whenever an SEI $\mathcal{I}$ is semi-unifiable the SEI rewriting system of Figure 9 will terminate with a proper normal form SEI $\mathcal{I}'$ (not equal to $\Box$) from which we can read off a most general semi-unifier S of $\mathcal{I}'$. As a result of Proposition 13 the substitutions S and $S|_{FV(\mathcal{I})}$ are most general semi-unifiers of $\mathcal{I}$. If $\mathcal{I}$ is not semi-unifiable then the rewriting system either stops with the "improper" SEI $\Box$ or it does not terminate.

The main reason for nontermination is that Rule 10 introduces new variables every time it is executed. Replacing it with the deceivingly pleasing rule [Pur87]

> $f(M_1, \ldots, M_k) \leq x$:
> Add the equation $x = f(M_1, \ldots, M_k)$.

indeed eliminates the nontermination problem of rewriting derivations, but also its correctness. To see this, consider, for example, the system $\mathcal{I}_1$ consisting of the single inequation $f(g(y), g(y)) \stackrel{?}{\leq} f(x, g(g(y)))$. There is a derivation that would lead us to claim, incorrectly, that $\mathcal{I}_1$ has no semi-unifiers.

If we consider system $\mathcal{I}_0 = \{f(y, y) \stackrel{?}{\leq} x, x \stackrel{?}{\leq} y\}$ it is easy to see that it is unsolvable. This is caught by the *extended occurs check*, Rule 9.

5.2 Arrow Graph Rewriting System

Fast unification algorithms [Hue76,PW78,MM82,ASU86] use *term graphs*, a data structure that supports sharing of subexpressions, to eliminate the potentially exponential cost of copying terms and applying substitutions. We present *arrow graphs*, which are term graphs with additional structure to represent equations and inequations between terms, and translate the SEI rewriting system to an arrow graph rewriting system.

5.2.1 Arrow graphs

A *term graph* is a graph that represents sets of terms over a given alphabet $\mathcal{A} = (F, a)$ and set of variables V. It consists of a set of nodes, N, a subset of which is labeled with function symbols from F, and the rest of which is labeled with variables from V. If f is a function symbol with

arity k, $k \geq 0$, every node n labeled with f has exactly k *ordered children*; i.e., there are k directed *term edges* originating in n and labeled with the numbers 1 through k. The variable labeled nodes have no children, and for every variable x there is at most one node labeled with x. If the term edges contain no cycles we say the term graph is *acyclic*.

Every node in an acyclic term graph can be interpreted as a term; for example, if n is a node labeled with function symbol f, and its children are $n_1, \ldots, n_k$ (in this order) representing terms $M_1, \ldots, M_k$, then n represents the term $f(M_1, \ldots, M_k)$. Note that for every set of terms there is an easily constructed, but generally non-unique term graph such that every term is represented in it.

An *arrow graph* is a term graph with an equivalence relation $\sim$ on its nodes representing equations and additional directed edges called *arrows*. Arrows are directed edges labeled by natural numbers, which indicate from which inequation in a given system of equations and inequations an arrow is derived. We call the labels of arrows *colors*, and we write $n_1 \stackrel{i}{\rightarrow} n_2$ if there is an i-colored arrow pointing from n_1 to n_2.

An *arrow graph representation* of a system of equations and inequations

$$\mathcal{I} = \{M_0 \stackrel{?}{=} N_0, M_1 \stackrel{?}{\leq} N_1, \ldots, M_k \stackrel{?}{\leq} N_k\}$$

is a term graph with (not necessarily distinct) nodes $m_0, m_1, \ldots, m_k, n_0, n_1, \ldots, n_k$ representing the terms in $\mathcal{I}$, and arrows from m_i (representing M_i) to n_i (representing N_i) for $1 \leq i \leq k$ that have pairwise distinct colors, the only nontrivial equivalence being $m_0 \sim n_0$. To summarize, terms are represented by nodes in the underlying term graph, equations by an equivalence relation, and inequations by directed, labeled edges, called colored arrows.

5.2.2 Algorithm A

Algorithm A is given in Figure 10. It operates as follows. A set of closure conditions on arrow graphs, depicted in Figure 11, are interpreted as rewrite rules. It repeatedly rewrites an arrow graph by nondeterministically choosing an applicable rewrite rule until no rewrite rule is applicable any more. In the final, *normalized* arrow graph every node can be interpreted as a unique term. Algorithm A is correct since the arrow graph rewriting steps correspond directly to rewriting steps in the SEI rewriting system of Figure 9.

5.3 Implementing Type Inference

Algorithm A is a very flexible, practical basis for implementing a type inferencer for a language with polymorphic recursion. In particular it can easily be adapted to work in both a batch-oriented and a highly interactive programming environment. Since it is not tied to the syntax or peculiarities of any given programming language it may even serve as a generic basis for several languages.

We envisage a polymorphic type inferencer to have two main components that, at least conceptually, execute concurrently: A *constraint extraction module* and a *semi-unification module*. The constraint extraction module generates equations and inequations from the abstract syntax tree and symbol table of a front end input processor and feeds them to the semi-unification module as (parts of) arrow graph representations; the semi-unification module normalizes them using Algorithm A (Figure 10). In a batch-oriented programming environment all the typing constraints of a complete program would be generated before they are fed to the semi-unification module. In an interactive environment the constraints — such as for a single function definition

Let G be an arrow graph. Apply the following steps (depicted also in Figure 11) until convergence:

1. If there exist nodes m and n labeled with a function symbol f and with children m_1, m_2 and n_1, n_2, respectively, such that $m \sim n$ then merge the equivalence classes of m_1 and n_1 and of m_2 and n_2.
2. If there exist nodes m and n labeled with a function symbol f and with children m_1, m_2 and n_1, n_2, respectively, such that $m \xrightarrow{i} n$ then place arrows $m_1 \xrightarrow{i} n_1$ and $m_2 \xrightarrow{i} n_2$.
3. If there exist nodes m_1, m_2, n_1, and n_2 such that
   (a) $m_1 \sim n_1$, $m_1 \xrightarrow{i} m_2$ and $n_1 \xrightarrow{i} n_2$ then merge the equivalence classes of m_2 and n_2;
   (b) $m_1 \sim n_1$, $m_1 \xrightarrow{i} m_2$ and $m_2 \sim n_2$ then place an arrow $n_1 \xrightarrow{i} n_2$.
4. (a) (Extended occurs check) If there is a path consisting of arrows of any color (arrow path) from n_1 to n_2 and n_2 is reachable from n_1 via (more than zero) term edges, then reduce to the improper arrow graph $\Box$.
   (b) If the extended occurs check is *not* applicable and there exist nodes m and n such that m is labeled with function symbol f and has children m_1, m_2, n is not equivalent to a function symbol labeled node, and there is an arrow $m \xrightarrow{i} n$ then: create new nodes n', n'_1, n'_2 (each initially in their own equivalence class); label n' with function symbol f; label n'_1 and n'_2 with new variables x' and x'', respectively; make n'_1, n'_2 the children of n'; and merge the equivalence classes of n and n'.

Figure 10: Algorithm A

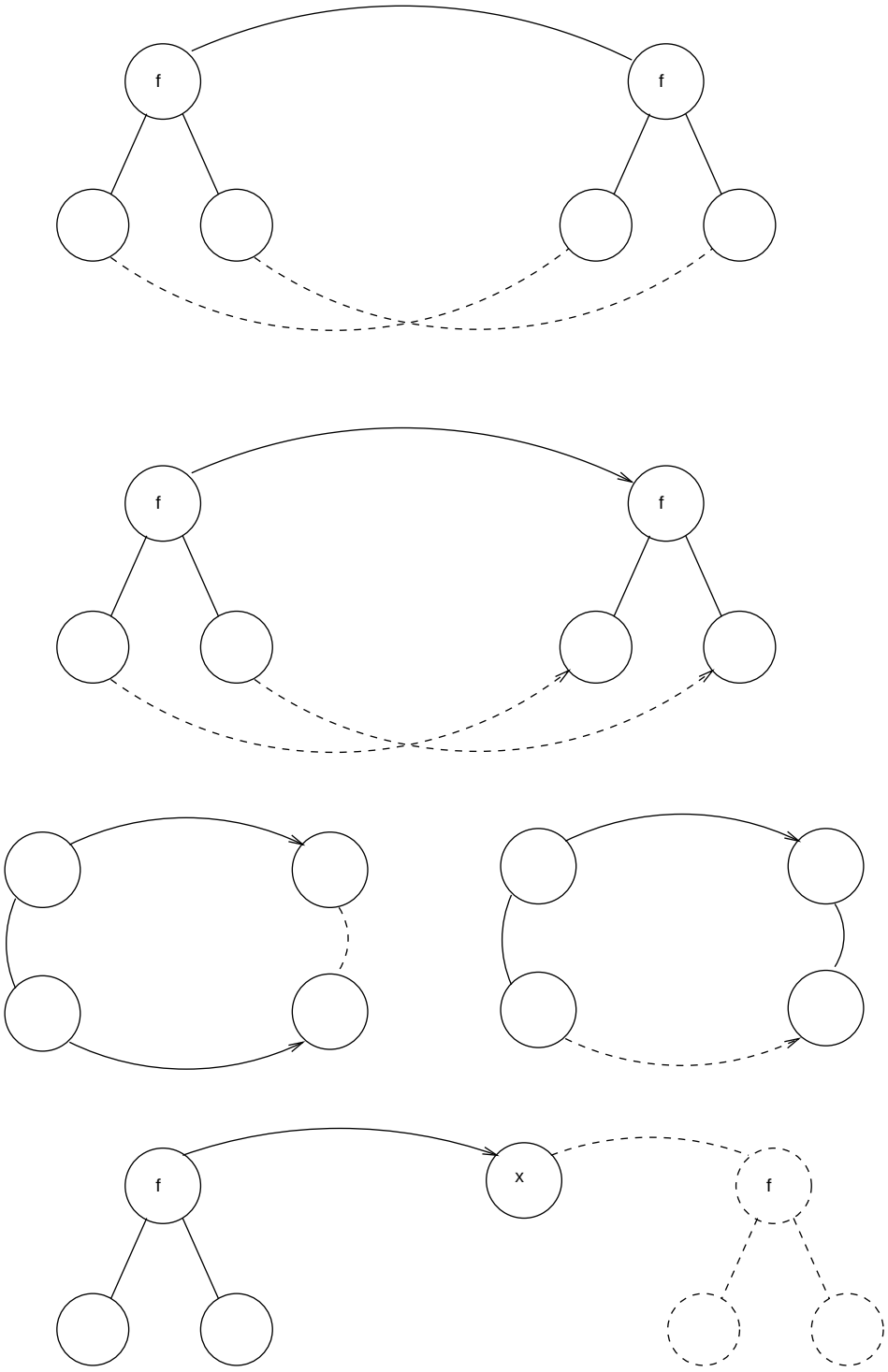

Figure 11: Closure rules

— could be passed to the semi-unifier for *incremental* type checking. In the case when typing constraints are given to the semi-unifier as soon as they become available from the input processor, the type checker runs essentially completely synchronously with input processing, thus displaying a high degree of "interactivity".

The above scheme is simple and works well interactively as long as typing constraints are only added and not removed during input processing. To facilitate flexible program development a type inferencer will also have to support incremental type checking where constraints are *eliminated* due to editing changes in the underlying program, which may actually come from the detection of a type error in an interactive program development session.

We have implemented the functional semi-unification algorithm described in Henglein [Hen88] in SETL [SDDS86] to experiment with semi-unification, which has proved very helpful in the beginning stages of our work on type inference and semi-unification. There are currently no implementations of type inferencers for realistic languages that integrate the ideas presented above, however.

6 Size-Bounded Typing

In view of its undecidability type inference for languages with polymorphic recursion would appear to be hopeless and futile. Interestingly, even without polymorphic recursion type inference is, theoretically, intractable: ML typing is DEXPTIME-complete [KM89,Mai90,KMM91, KTU90a]. This is in remarkable contrast to the overall positive practical experience with the actual use of languages based on ML's typing rules.

In Section 6.1 we offer some general considerations to suggest that the observed practicality of polymorphic type inference is not merely coincidental, and in Section 6.2 we briefly formalize some of our considerations. The statements and results in this section should be taken with a grain of salt. They reflect our concern for explaining and reconciling the apparent contradictions of theory (staggering lower bounds) and practice (acceptable performance of actual implementations), and it is hoped that they provide fruitful insights leading to more substantial work on this question.

6.1 Theoretical Intractability and Practical Utility of Polymorphic Type Inference

Practical experience suggests that the complexity-theoretic cost of polymorphic type inference in the Damas-Milner and the Milner-Mycroft type systems is too pessimistic. Consider the following points. First, *all* possible typings of an expression can be represented by its principal typing alone. Second, due to the principal typing property, there are relatively simple type inference algorithms that do not necessitate any backtracking or other complicated control mechanisms. Third, languages such as ML, Miranda, and ABC have been in use for several years now, and the type inferencers in these systems have been reasonably efficient in actual use. In fact their observed efficiency has helped promulgate the myth that ML type checking is *theoretically* efficient since it was believed, for almost ten years, to have a worst-case polynomial running time of low degree.

The sole reason why the cost of type inference in the Milner-Mycroft Calculus or the Damas-Milner Calculus can get out of hand is because the type inferencer has to manipulate type expressions whose sizes are inconceivably much bigger than the underlying (untyped) program. A conventional remedy for eliminating problems with type inference is to mandate explicit, fully

typed declarations of variables, parameters and other basic syntactic units. Applying this sort of remedy to the Milner-Mycroft Calculus or even ML highlights, though, why type *checking* (with explicit type information embedded in the program) is no more "practical" than type *inference* (with no or only optional type information in the program): there are constructible ML-programs that fit on a page and are, at least theoretically, well-typed, yet writing (deriviations of) their principal types would require more than the number of atoms in the universe. So writing an explicitly typed version of the program is manifestly impossible to start with. More provocatively we might say that in this case type inference is actually faster than type checking since an implemented type inferencer can infer types faster than a programmer can write them down.

The formalization of type inference as a combinatorial problem in an inference system — such as the Damas-Milner or the Milner-Mycroft Calculus — does not take the *intensional* character of types and typings into account. In the *semantic* world types and typings are generally viewed as *abstractions* of the *behavior* of programs and their parts. We argue that this is reflected in the syntactic world: type expressions are used and thought of as *syntactic abstractions* of the syntactic parts of programs.[13] If the size of a type expression stands in no reasonable relation to the size of the underlying program text — say it is exponentially bigger — then it is unreasonable to consider the type expression an abstraction of the program text. This can be interpreted as saying that a program has no "reasonable" type description of its behavior in the given typing discipline: it should be considered type incorrect. The rationale behind this decision could be formulated as "(Good) programs have small types".[14] A similar argument has been suggested by Boehm [Boe89], and the observations about ML programs made in [KM89] are consistent with our explanation.

This is not to suggest that *imposing* a bound on the sizes of types in a type system is a good *definitional requirement* on type systems but that a "small" size bound is a good *property* of a type system. It remains to be seen whether such a type system can be defined in a logically robust fashion. One possibility is to require type declarations for *some* bindings, but not all. For example, Hope mandates explicit type declarations for recursive definitions. This makes Hope type inference no harder than ML type inference. If, additionally, type declarations are mandatory for nonrecursive (**let**-) definitions then type inference is no harder than unification.

6.2 Type Inference for Programs with Small Types

A *typed λ-expression* e is an expression e' in which every λ-binding is decorated with a type and every **let**- and **fix**-binding is decorated with a type scheme; e' is *well-typed* if there is a typing derivation in the Milner-Mycroft Calculus with these particular type assumptions on variables. We say e is the (unique) *untyped version* of e', and e' is a *typed version* of e. If e' is well-typed, then we call e' a *well-typed version* of e. Clearly, if e' is well-typed then e is typable.

Let us say an (untyped) expression e of size n has a *small typing* if it has a well-typed version e' that is at most of size $p(n)$ for a fixed polynomial p.

Theorem 8 *Milner-Mycroft typability with small types is polynomial-time decidable.*

Proof: The size of any well-typed version of e must be at least as big as the size of the normalized arrow graph representation of the equations and inequations constructed from e (see

[13] This reflects a view in which types are *not* first-class objects in a programming language.

[14] "And bad programs *fail* with small types."

Section 5.2). Since the arrow graph representation monotonically grows (after a polynomial number of rewriting steps) only a polynomial number of rewriting steps can be executed before the algorithm terminates or the arrow graph representation becomes bigger than $p(n)$. ■

If we consider, in any given type system, a well-typed version e' of an untyped program e as a *witness* to the fact that e is well-typed, then any typing problem whose witnesses are required to be small (polynomial-sized) is in *NP* as long as type checking explicitly typed programs can be done in polynomial time. Type checking in the Damas-Milner, the Milner-Mycroft, and the higher-order typed λ-calculi $F_2, F_3, \ldots, F_\omega$, as well as type checking in the presence of Ada-style overloading can be done in polynomial time. Their associated type inference problems are all intractable [Mai90,KTU90b,HM91] [ASU86, exercise 6.25]. Note, however, that type inference with small types for the Damas-Milner and Milner-Mycroft systems are in *P* whereas type inference with overload resolution remains *NP*-complete. Also, we conjecture that higher-order typability with small types is *NP*-complete. This lends some technical expression to the intuition that "overload resolution" as above is much harder than polymorphic type inference.

7 Summary and Outlook

The Milner-Mycroft Calculus is a type system that permits polymorphic usage of recursively defined functions inside their own definitions. This extension does away with the need to carefully craft a set of function definitions into a list of nested definitions in order to satisfy type checking constraints inherent in ML-style type systems. We have shown that the type inference problem for the Milner-Mycroft Calculus is *log*-space equivalent to semi-unification. We have presented a semi-unification algorithm that can be used as a generic basis for a batch-oriented or interactive type checker/inferencer. Even though semi-unification was recently shown to be undecidable we have argued that, in practice, programs have "small types", if they are well-typed at all, and Milner-Mycroft type inference for small types is tractable. This, we think, also provides insight into why ML type checking is usable and used in practice despite its theoretical intractability.

The utility of polymorphic recursion and polymorphic function arguments still remain to be evaluated in a concrete language setting. Special attention has to be given to the interaction of polymorphic type inference with abstract data types, coercions, inheritance, and overloading.

Acknowledgements

I wish to thank Ed Schonberg and Bob Paige for their support and their valuable interaction about the algorithmic aspects of type inference. I am especially thankful to Harry Mairson for taking a great interest in this paper and suggesting many improvements in its exposition. Finally, I have benefited greatly from Hans Leiß' insights into semi-unification and its special cases.

References

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986. Addison-Wesley, 1986, Reprinted with corrections, March 1988.

[Bar84] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[BMS80] R. Burstall, D. MacQueen, and D. Sannella. Hope: An experimental applicative language. In *Stanford LISP Conference 1980*, pages 136–143, 1980.

[Boe89] H. Boehm. Type inference in the presence of type abstraction. In *Proc. SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 192–206. ACM, ACM Press, June 1989.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation (Information and Control)*, 76:138–164, 1988.

[CDDK86] D. Clement, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. *INRIA Centre Sophia Antipolis*, RR No. 529, May 1986.

[CF58] H. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, 1958.

[Cho86] C. Chou. Relaxation processes: Theory, case studies and applications. Master's thesis, UCLA, February 1986. Technical Report CSD-860057.

[CHS72] H. Curry, J. Hindley, and J. Seldin. *Combinatory Logic*, volume II of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1972.

[Cur69] H. Curry. Modified basic functionality in combinatory logic. *Dialectica*, 23:83–92, 1969.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, Dec. 1985.

[Dam84] L. Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1984. Technical Report CST-33-85 (1985).

[DKM84] C. Dwork, P. Kanellakis, and J. Mitchell. On the sequential nature of unification. *J. Logic Programming*, 1:35–50, 1984.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[DR90] J. Dörre and W. Rounds. On subsumption and semiunification in feature algebras. In *Proc. 1990 IEEE Symp. on Logic in Computer Science (LICS)*, pages 300–311. IEEE Computer Society Press, July 1990.

[Ede85] E. Eder. Properties of substitutions and unifications. *J. Symbolic Computation*, 1:31–46, 1985.

[Gir71] J. Girard. Une extension de l'interpretation de Godel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symp.*, pages 63–92, 1971.

[GLT89] J. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[GMP90] L. Geurts, L. Meertens, and S. Pemberton. *The ABC Programmer's Handbook.* Prentice Hall, New York, 1990.

[GRDR88] P. Giannini and S. Ronchi Della Rocca. Characterization of typings in polymorphic type discipline. In *Proc. Symp. on Logic in Computer Sciene*, pages 61–70. IEEE, Computer Society, Computer Society Press, June 1988.

[Hen88] F. Henglein. Type inference and semi-unification. In *Proc. ACM Conf. on LISP and Functional Programming (LFP), Snowbird, Utah*, pages 184–197. ACM Press, July 1988.

[Hen89] F. Henglein. *Polymorphic Type Inference and Semi-Unification.* PhD thesis, Rutgers University, April 1989. Available as NYU Technical Report 443, May 1989, from New York University, Courant Institute of Mathematical Sciences, Department of Computer Science, 251 Mercer St., New York, N.Y. 10012, USA.

[Hen90] F. Henglein. Fast left-linear semi-unification. In *Proc. Int'l. Conf. on Computing and Information*, pages 82–91. Springer, May 1990. Lecture Notes of Computer Science, Vol. 468.

[Her68] J. Herbrand. Recherches sur la theorie de la demonstration. In *Ecrits logiques de Jacques Herbrand.* PUF, Paris, 1968. thèse de Doctorat d'Etat, Université de Paris (1930).

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, Dec. 1969.

[HM91] F. Henglein and H. Mairson. The complexity of type inference for higher-order typed lambda calculi. In *Proc. 18th ACM Symp. on Principles of Programming Languages (POPL), Orlando, Florida*, pages 119–130. ACM Press, Jan. 1991.

[Hoo65] P. Hooper. *The Undecidability of the Turing Machine Immortality Problem.* PhD thesis, Harvard University, June 1965. Computation Laboratory Report BL-38; also in Journal of Symbolic Logic, 1966.

[HS86] R. Hindley and J. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press, 1986.

[Hue76] G. Huet. *Résolution d'equations dans des langages d'ordre 1, 2, ..., omega (thèse de Doctorat d'Etat).* PhD thesis, Univ. Paris VII, Sept. 1976.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. Assoc. Comput. Mach.*, 27(4):797–821, Oct. 1980.

[HW90] P. Hudak and P. (editors) Wadler. *Report on the Programming Language Haskell*, April 1990.

[JW86] G. Johnson and J. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *Proc. 13th Annual ACM Symp. on Principles of Programming Languages*, pages 44–57. ACM, Jan. 1986.

[KM89] P. Kanellakis and J. Mitchell. Polymorphic unification and ML typing (extended abstract). In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*. ACM, January 1989.

[KMM91] P. Kanellakis, H. Mairson, and J. Mitchell. Unification and ML type reconstruction. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic — Essays in Honor of Alan Robinson*. MIT Press, 1991.

[KMNS91] D. Kapur, D. Musser, P. Narendran, and J. Stillman. Semi-unification. *Theoretical Computer Science*, 81(2):169–188, April 1991. Based on paper presented at Conf. on Foundations of Software Technology and Teoretical Computer Science (FST-TCS), Dec. '88, Springer Lecture Notes in Computer Science, Vol. 338.

[KT90] A. Kfoury and J. Tiuryn. Type reconstruction in finite-rank fragments of the polymorphic λ-calculus. In *Proc. 5th Annual IEEE Symp. on Logic in Computer Science (LICS), Philadelphia, Pennsylvania*, pages 2–11. IEEE Computer Society Press, June 1990.

[KTU88] A. Kfoury, J. Tiuryn, and P. Urzyczyn. A proper extension of ML with an effective type-assignment. In *Proc. 15th Annual ACM Symp. on Principles of Programming Languages*, pages 58–69. ACM, ACM Press, Jan. 1988.

[KTU89] A. Kfoury, J. Tiuryn, and P. Urzyczyn. Computational consequences and partial solutions of a generalized unification problem. In *Proc. 4th IEEE Symposium on Logic in Computer Science (LICS)*, June 1989.

[KTU90a] A. Kfoury, J. Tiuryn, and P. Urzyczyn. ML typability is DEXPTIME-complete. In *Proc. 15th Coll. on Trees in Algebra and Programming (CAAP), Copenhagen, Denmark*, pages 206–220. Springer, May 1990. Lecture Notes in Computer Science, Vol. 431.

[KTU90b] A. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. In *Proc. 22nd Annual ACM Symp. on Theory of Computation (STOC), Baltimore, Maryland*, pages 468–476, May 1990.

[Lei83] D. Leivant. Polymorphic type inference. In *Proc. 10th ACM Symp. on Principles of Programming Languages*, pages 88–98. ACM, Jan. 1983.

[Lei87] H. Leiß. On type inference for object-oriented programming languages. In *Proc. 1st Workshop on Computer Science Logic*. Springer-Verlag, Lecture Notes Computer Science, Vol 329, Oct. 1987.

[Lei89a] H. Leiß. Decidability of semi-unification in two variables. Technical Report INF-2-ASE-9-89, Siemens, Munich, Germany, July 1989.

[Lei89b] H. Leiß. Semi-unification and type inference for polymorphic recursion. Technical Report INF2-ASE-5-89, Siemens, Munich, Germany, 1989.

[LH91] H. Leiß and F. Henglein. A decidable case of the semi-unification problem. In *Proc. 16th Int'l Symp. on Mathematical Foundations of Computer Science (MFCS), Poland*. Springer, Sept. 1991. Lecture Notes in Computer Science, Vol. 520.

[LMM87] J. Lassez, M. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kauffman, 1987.

[Mai90] H. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Proc. 17th ACM Symp. on Principles of Programming Languages (POPL)*. ACM, Jan. 1990.

[Mee83] L. Meertens. Incremental polymorphic type checking in B. In *Proc. 10th ACM Symp. on Principles of Programming Languages (POPL)*, pages 265–275, 1983.

[MH88] J. Mitchell and R. Harper. The essence of ML. In *Proc. 15th ACM Symp. on Principles of Programming Languages (POPL)*. ACM, Jan. 1988.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[Mit88] J. Mitchell. Polymorphic type inference and containment. *Information and Control*, 76:211–249, 1988.

[Mit90] J. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North-Holland, 1990.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, Apr. 1982.

[MO84] A. Mycroft and R. O'Keefe. A polymorphic type system for PROLOG. *Artificial Intelligence*, 23:295–307, 1984.

[Mor68] J. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, MIT, 1968.

[MTH90] R. Milner, M. Tofte., and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proc. 6th Int. Conf. on Programming, LNCS 167*, 1984.

[Pud88] P. Pudlák. On a unification problem related to Kreisel's conjecture. *Commentationes Mathematicae Universitatis Carolinae*, 29(3):551–556, 1988.

[Pur87] P. Purdom. Detecting looping simplifications. In *Proc. 2nd Conf. on Rewrite Rule Theory and Applications (RTA)*, pages 54–62. Springer-Verlag, May 1987.

[PW78] M. Paterson and M. Wegman. Linear unification. *J. Computer and System Sciences*, 16:158–167, 1978.

[Rey74] J. Reynolds. Towards a theory of type structure. In *Proc. Programming Symposium*, volume 19 of *LNCS*, pages 408–425. Springer-Verlag, 1974.

[SDDS86] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[SS86] L. Sterling and E. Shapiro. *The Art of PROLOG*. MIT Press, 1986.

[Tur86] D. Turner. An overview of Miranda. *SIGPLAN Notices*, 21(12):158–166, Dec. 1986.

[Wan86] M. Wand. Finding the source of type errors. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 38–43. IEEE, June 1986.