On Polymorphic Type Inference and Semi-Unification: Algebraic, Computational, and Conceptual Results (Technical Summary)*

Fritz Henglein†
Department of Computer Science
Rijksuniversiteit Utrecht
PO Box 80.089
3508 TB Utrecht
The Netherlands
Internet: henglein@cs.ruu.nl

July 26, 1989

1 Introduction and Summary

The advent of ML [GMM+78, **?**] has sparked the development of programming languages that try to combine the flexibility and conciseness of largely declaration-free, dynamically typed languages, such as LISP, with the safety and implementation efficiency of statically typed languages, such as Pascal. Other languages that also use *polymorphic* typing rules are SPS [Wan84], Miranda [Tur86], Haskell. At the heart of these languages is the combination of a powerful, flexible type system with reliance on, to a large degree, compile-time type inference. This enables the user to write concise code in the style of type-free languages, and yet have it checked at compile time for possible type errors.

*Submitted to European Symposium on Programming (ESOP) '90, May 15-18, 1990

†This research was conducted while the author was at New York University. It has been supported by the ONR under contract numbers N00014-85-K-0413 and N00014-87-K-0461.

The success of ML's type system may be attributed mainly to two factors. First, for every type correct ML program there is a "canonical" (or "principal") way of associating type information with every variable occurrence and expression in the program (*principal typing property*). Second, there is a practical unification-based type inference algorithm that is sound and complete with respect to the type system [Mil78]; in particular, it effectively computes the canonical type information, which could be used for program documentation and optimization.

There is a significant problem with applying ML's type system to languages without nesting of definitions (of functions, procedures or other objects). It has to do with a peculiarity in the typing rule for recursive definitions in ML: Occurrences of a recursively defined function in ML *inside* its definition body can only be used *monomorphically* (all of them 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). Logic programs can be viewed as massive mutually recursive definitions where *all* the antecedents of the constituent Horn clauses represent the bodies. As observed in [MO84], an ML-style type system for a logic programming language eliminates almost all polymorphism since *all* occurrences of a predicate in the antecedents are required to have literally identical types! More generally, languages that do not permit nested scoping of definitions (e.g., Prolog [SS86], SETL [SDDS86], ABC [MP85][1]) treat definitions as generally mutually recursive and suffer the same fate in an ML-style type system.

For this reason, Meertens [Mee83] and Mycroft

[1] ABC used to be called B.

[Myc84] extended ML's monomorphic typing rule for recursive definitions to a more general *polymorphic* typing rule. This extended type inference system has already been studied by Wadsworth in the late 70's [Mac88] and also, more recently, by Leiß [Lei87], Kfoury *et al.* [KTU88b, KTU88a, KTU89], and this author [Hen88]. It permits polymorphic usage of recursively defined functions (or procedures) *everywhere* and thus "solves" the peculiarity associated with ML's recursion typing rule.

We call the formal type inference system associated with ML the *Damas-Milner Calculus* and its extension with polymorphic recursion *Milner-Mycroft Calculus*. A simpler type system without any polymorphic usage of variables is the *Curry-Hindley Calculus*. All three type inference systems are presented in Appendix A.

Type inference in the Milner-Mycroft Calculus is intimately connected with *semi-unification*, an "asymmetric", generalized form of unification (see section 2). Definitions of semi-unification, system of inequalities, semi-unifiers and a powerful characterization theorem connecting type inference and semi-unification can be found in section 2. Furthermore, in this paper we present the following new results.

- We show that the set of derivable typings in the Milner-Mycroft Calculus, the Damas-Milner Calculus, and the Curry-Hindley Calculus form a complete lattice under a natural ordering on typings. This generalizes the principal typing properties for these type inference systems proved in [Hin69] (for combinatory logic), [BY79] (for λ-calculus), [DM82], and [Myc84]. These generalized principal typing properties follow directly from the connection of type inference with semi-unification and the following result presented here: For any system of inequalities the set of all semi-unifiers forms a complete lattice (but not a filter) under weak equivalence.[2] This result is of independent interest as it proves the existence of most general semi-unifiers and generalizes an unpublished result of Baaz (see references in [Pud88]).

- General semi-unification is not known to be decidable or undecidable[3]. We prove that *left-linear* semi-unification (see section 3) is polynomial-time decidable by employing a modified dynamic transitive closure algorithm. This improves a recent exponential upper bound given in [KTU89]. We briefly treat other special cases of semi-unification.

[2] This terminology is defined in section 3.

[3] Kfoury, Tiuryn, and Urzyczyn have recently announced a proof of undecidability ("types" mailing list, September 25th, 1989)

- Despite its observed practicality ML typing has been shown to be PSPACE-hard [KM89][4]. To help explain this discrepancy we argue that "real programs have small types" and show that, under this premise, type inference is tractable — i.e., polynomial-time decidable — in the Milner Calculus *and* in the Milner-Mycroft Calculus. This lends technical expression to observations made in [KM89] and [Boe89] and justifies using a potentially prohibitively expensive type checker in a compiler.

Our results may be interpreted as follows. The Milner-Mycroft Calculus is a very desirable polymorphic type discipline since it permits polymorphic usage uniformly inside and outside of recursive definitions (in contrast to the Milner Calculus), yet it is still "well-behaved" in the sense that it has a (generalized) principal typing property (in contrast to higher order polymorphic extensions based on the Second Order λ-Calculus [Gir71, Rey74]). Type inference in the type systems under consideration here is a form of semi-unification, and vice versa[5]. Whereas not even the decidability of general semi-unification is known, the rather broad class of left-linear semi-unification problems can be solved in polyomial time, and under the premise that programs have "small" types (polynomial in the size of the untyped program) semi-unification — and thus type inference — is provably tractable. Since this holds for the Damas-Milner Calculus *and* for the Milner-Mycroft Calculus, our results offer justification, generally, for relying on automatic type inference in polymorphic programming languages and, specifically, for using the polymorphic typing rule for recursive definitions instead of ML's more restrictive monomorphic rule. An implementation of the polymorphic typing rule in the type checker of Standard ML of New Jersey is planned for the near future.

2 Polymorphic Type Inference and Semi-Unification

For the purpose of studying polymorphic type inference in isolation from other typing concerns, we shall restrict ourselves to a notationally mini-

[4]Recently, Kfoury, Tiuryn, and Urzyczyn have announced a proof of DEXPTIME-completeness for ML typing (types mailing list, September 22nd, 1989).

[5]The exact connection between polymorphic type inference and semi-unification is detailed in [Hen89].

mal programming language. As usual, we define typing displines by inference systems over typings (see Appendix A). The Curry-Hindley Calculus permits no definition of polymorphic functions; the Damas-Milner Calculus corresponds to the type system of (the functional core of) ML; and the Milner-Mycroft Calculus is the extension of the Damas-Milner Calculus with a polymorphic typing rule for recursive definitions.

Semi-unification is a problem akin to unification. The preordering $\leq$ of *subsumption* on first-order terms is defined by $M \leq N$ if there exists a substitution ρ such that $\rho(M) = N$. A system $\{M_{11} = M_{12}, \ldots, M_{k1} = M_{k2}, N_{11} \leq N_{12}, \ldots, N_{l1} \leq N_{l2}\}$ of (term) equations and (term subsumption) inequalities is *semi-unifiable* if there is a substitution σ such that all the equalities and subsumption statements $\sigma(M_{11}) = \sigma(M_{12}), \ldots, \sigma(M_{k1}) = \sigma(M_{k2}), \sigma(N_{11}) \leq \sigma(N_{12}), \ldots, \sigma(N_{l1}) \leq \sigma(N_{l2})$ hold. The equations are superfluous since they can be encoded by inequalities.[6] Consequently we shall usually only talk of a system of inequalities or a system of equations. *Polymorphic unification*, an extension of ordinary unification recently used by Kanellakis and Mitchell to prove type checking in ML PSPACE-hard [KM89], defines a subclass of semi-unification problems.

The significance of semi-unification for polymorphic type inference lies in the following theorem.

Theorem 1 *The following three problems are polynomial-time equivalent:*

1. *typability in the Milner-Mycroft Calculus;*
2. *(nonuniform) semi-unifiability;*
3. *typability in the Milner-Mycroft Calculus restricted to expressions of the form* **fix f.e** *where e is a* **fix**- *and* **let**-*free* λ-*expression.*

The step (1) $\rightarrow$ (2) was shown in [Hen88]. Step (2) $\rightarrow$ (3) was proved independently in [KTU89] and [Hen89]. Step (3) $\rightarrow$ (1) is trivial. This characterization was extended to the Second Order Lambda Calculus of bounded rank 2 in [KTU89] (for rank-bounded typing see [Lei83]).

In [Hen89] it is also shown that not only the decision problems above are preserved in the steps (1) $\rightarrow$ (2) and (2) $\rightarrow$ (3), but also the algebraic structure of solutions to these problems. This is exploited in the proof of the generalized principal typing property in section 3.

[6] For example, $M_1 = M_2$ has a unifier if and only if $f(x, x) \leq f(M_1, M_2)$ has a semi-unifier, where x is a variable not occurring in M_1, M_2.

3 New Results

As already indicated in section 1 the new results we report are divided into three sections:

- Algebraic properties of semi-unification and derivable typings (generalized principal typing properties);
- algorithms for special cases of semi-unification and their computational complexity;
- type inference for programs with small types.

3.1 Algebraic Properties of Semi-Unification and Principal Typing Properties

In this subsection we present the main structure theorem for semi-unifiers and its corollaries, the generalized principal typing properties for the Curry-Hindley, Damas-Milner, and Milner-Mycroft type inference systems. To emphasize some similarities and differences of semi-unification with unification we also recast some known results on the algebraic structure of unifiers.

It is well-known that unification problems have *most general unifiers* that are unique modulo "renaming of variables". There are, however, several different equivalence relations on substitutions that formalize what exactly "renaming of variables" means.[CL73, SS86, LMM87]

A substitution σ_1 is *at least as general* as σ_2, written as $\sigma_1 \leq \sigma_2$, if there is a substitution ρ such that $\rho \circ \sigma_1 \equiv \sigma_2$; σ_1 and σ_2 are *strongly equivalent*, $\sigma_1 \cong \sigma_2$, if $\sigma_1 \leq \sigma_2$ and $\sigma_2 \leq \sigma_1$. A somewhat "coarser" formalization results from considering only a subset of variables in the definition of generality above. Let W be a subset of V, the set of all variables. Substitution σ_1 is *at least as general* as σ_2 *w. r. t.* W, written as $\sigma_1 \leq_W \sigma_2$, if there is a substitution ρ such that $\rho(\sigma_1(x)) = \sigma_2(x)$ for all $x \in W$; σ_1 and σ_2 are *weakly equivalent w. r. t.* the system of inequalities (or equations) S, $\sigma_1 \cong_{V(S)} \sigma_2$, if $\sigma_1 \leq_{V(S)} \sigma_2$ and $\sigma_2 \leq_{V(S)} \sigma_1$ where $V(S)$ is the set of all variables occurring in S.

The most widely used notion for "renaming of variables" is *strong equivalence* since it is independent of particular variable sets. It is well-known that every unification problem (i.e., term equation) that has a unifier at all has a most general unifier; i.e., there is a unifier σ such that for all unifiers σ' we have $\sigma \leq \sigma'$ (c.f., [LMM87]). Since $\leq$ is a preorder this implies that

most general unifiers are unique modulo strong equivalence. Furthermore, most general unifiers are also unique modulo weak equivalence even though not every most general unifier modulo weak equivalence is a most general unifier modulo strong equivalence.

Apparently in analogy to unification it has been stated that systems of inequalities have *most general semi-unifiers* that are unique modulo strong equivalence [Cho86, PM88]. We show that systems of inequalities that have semi-unifiers at all have most general semi-unifiers modulo *weak* equivalence, but, in general, *not* with respect to *strong* equivalence.

A substitution σ is idempotent if $\sigma = \sigma \circ \sigma$. Henceforth we shall assume the existence of an "undefined" (idempotent) substitution ω with $\sigma \leq \omega$ for all σ. Let us denote the set of all idempotent substitutions, with ω, by $\mathcal{IS}^{\omega}$. We can factor out the equivalence relation $\cong$ (or $\cong_W$) from a set Σ of substitutions by writing $\Sigma/_{\cong}$ (or $\Sigma/_{\cong}$, in which case $\leq$ (respectively $\leq_W$) canonically induces a partial order. Finally, for a system of inequalities S we write $\mathbf{SU}(S)$ for the set of all semi-unifiers of S; similarly for a system of equations S, we write $\mathbf{U}(S)$ for the set of unifiers of S.

Theorem 2 *(Eder [Ede85])*

1. *Every system of equations S has a most general unifier that is idempotent, and for every idempotent substitution σ there is a system of equations S' such that σ is a most general unifier of S' (with respect to $\leq$).*

2. *$((\mathcal{IS}^{\omega} \cap \mathbf{U}(S))/_{\cong_V}, \leq_V)$ is a complete lattice for every system of equations S.*

This theorem fails for semi-unification in a major way. In particular, we have that, with respect to $\leq$:

1. for any system of inequalities or equations S neither $\mathbf{U}(S)$ nor $\mathbf{SU}(S)$ induce a lower or upper semi-lattice modulo strong equivalence.

2. there are systems of inequalities that have a most general semi-unifier, but no idempotent one;

3. there are systems of inequalities with no most general semi-unifier.

The break-down of the structure of semi-unifiers under strong equivalence indicates that strong equivalence is *too* strong a formalization for

the intuitive notion of "renaming". By using weak equivalence much of the structure of unifiers carries over to semi-unifiers. In particular, the following theorem shows that most general semi-unifiers exist and are unique modulo weak equivalence.

Theorem 3 *1. Every system of inequalities S has a most general semi-unifier, and for every substitution σ there is a system of inequalities S' such that σ is a most general semi-unifier of S.*

2. $(\mathbf{SU}(S)/_{\cong_{V(S)}}, \leq_{V(S)})$ is a complete lattice for every system of inequalities S.

The proof is composed of two main parts: First it is shown that $\leq_W$ is Noetherian (see Huet [Hue80]); then a construction is given that shows that every pair of semi-unifiers has a greatest lower bound semi-unifier. This is sufficient since every Noetherian lower semi-lattice is a complete lower semi-lattice, and every complete lower semi-lattice is automatically a complete lattice.

Apart from the fact that this main structure theorem of semi-unification holds only for weak equivalence, not strong equivalence, another structural difference between the lattice of unifiers in Eder's theorem and the lattice of semi-unifiers in this theorem should be pointed out: The lattice of unifiers in Eder's theorem is a *filter*; that is, it is the set of *all* (idempotent) substitutions[7] σ' such that $\sigma \leq \sigma'$ and σ is a most general unifier. In the lattice of semi-unifiers of a system of inequalities S in the above theorem, on the other hand, there may be substitutions σ' such that $\sigma \leq_{V(S)} \sigma'$ where σ is a most general semi-unifier of S, but σ' is *not* a semi-unifier of S.[8]

As a consequence of the connections of semi-unification and polymorphic type inference the above structure theorem yields a simultaneous algebraic proof of the principal typing properties of the Hindley Calculus [Cur69, Hin69, BY79], the Milner Calculus [DM82], and the Milner-Mycroft Calculus [Myc84]; in fact, the principal typing property can be strengthened by showing that all derivable typings for a given (typable) expression e form a complete lattice with respect to a natural ordering on typings.

[7] viewed as representatives of their strong equivalence classes

[8] This may be viewed informally as an indication that semi-unification algorithms can be expected to be structurally more complex than unification algorithms since it is not enough to encode a "current solution" space by a single substitution after processing part of the input; additionally the "holes" above the current most general semi-unifier need to be encoded.

A substitution S on monotypes can be applied to a polytype σ by simultaneously replacing only the free variables in σ while renaming bound type variables in σ to avoid capture of (necessarily free) type variables from S. Such a substitution can be extended to type assignments, $S(A)(x) = S(A(x)), x \in \mathbf{dom}\, A$ and to typings, $S(A \supset e : \sigma) = S(A) \supset e : S(\sigma)$. The notion of a *generic instance* of a polytype is defined inductively as follows.

1. If σ is a monotype, then σ is a generic instance of itself.
2. If σ is a polytype, $\sigma = \forall t.\sigma'$, τ is a generic instance of σ', and τ' is an arbitrary monotype then $\tau[\tau'/t]$ is a generic instance of σ

The *generic instance preordering* $\sqsubseteq$ on polytypes [DM82] is defined as follows: $\sigma_1 \sqsubseteq \sigma_2$ whenever every generic instance of σ_2 is also a generic instance of σ_1. This preordering can be extended to type assignments by $A \sqsubseteq A' \Leftrightarrow (\forall x \in \mathbf{dom} A)\ A(x) \sqsubseteq A'(x)$. Finally, we define the relation $(A \supset e : \sigma) \leq (A' \supset e' : \sigma')$: it holds if and only if there is a substitution S such that

1. $S(A) \sqsubseteq A'$,
2. $e = e'$,
3. $S(\sigma) \sqsubseteq \sigma'$.

Clearly $\leq$ defines a preorder that induces canonically a partial order, also denoted by $\leq$. Now we can formulate the following corollary of the main structure theorem of semi-unification.

Corollary 1 *Let X be "Curry-Hindley", "Damas-Milner", or "Milner-Mycroft". For any expression e that is typable in the X Calculus, the set of derivable typings for e in X forms a complete lattice under the partial order* $\leq$.

Of course, this corollary is only true if we assume the existence of a "nonsense" typing ω derivable for every e such that $T \leq \omega$ for all typings T. This corollary is a consequence of the reductions used in the proof of the characterization theorem in section 2 and the main structure theorem of semi-unification above.

3.2 Computational Complexity of Semi-Unification

Mycroft [Myc84] provided an elegant, but provably nonterminating algorithm for computing principal typings. Meertens [Mee83] described a more involved incremental type inference algorithm for ABC, which also computes principal typings and has nonterminating computations. Both Mycroft and Meertens left the computability of that question and the decidability of the Milner-Mycroft Calculus open. This question has turned into somewhat of a *fata morgana* since several attempts at proving decidability have been unsuccessful [Mee83, Lei87, KTU88b, Hen87].[9] Consequently the focus has shifted to special cases of semi-unification.

Uniform semi-unification — semi-unification with a single inequality — was shown to be (exponential-time) decidable independently in [Hen88], [Pud88], and [KMNS88]. If it is not desired that (a suitable representation of) most general semi-unifiers is computed, then uniform semi-unification is polynomial-time decidable [KMNS88]. Semi-unification with only two variables has recently been shown to be decidable [Lei89], too.

Another interesting class of semi-unification problems is *left-linear* semi-unification. An instance of this problem consists of a system of inequalities in which every term on the left-hand side of an inequality is *linear*; i.e., no variable occurs more than once in it. Kfoury *et al.* have proved that left-linear semi-unification is decidable [KTU89]. The running time of their algorithm is $\Omega(2^{n^2})$. We improve this result by showing

Theorem 4 *Left-linear semi-unification is polynomial-time decidable.*

A simple analysis of our algorithm gives a running time of $O(n^3)$. The bottleneck is a dynamic transitive closure computation. It seems likely that better analysis and exploitation of the specific structure of the graphs over which the transitive closure is computed may lead to an asymptotic improvement of the running time.[10] Note that it can be shown that semi-unification over any ranked alphabet can be encoded (in logarithmic space) by semi-unification over the alphabet that has only one binary functor and no other functors, in particular no constants. These encodings, however, do *not* preserve left-linearity, and thus it is important to remark that, nonetheless, our bound applies to left-linear semi-unification problems over *any* ranked alphabet. A proof sketch for this theorem can be found in Appendix B.

[9] As mentioned before, a proof of undecidability of semi-unification has been announced by Kfoury, Tiuryn, and Urzyczyn.

[10] This is currently being investigated.

Some simple "positive" and "negative" results on subclasses of semi-unification are summarized below. We say a system of inequalities S is

- right-linear, if every right-hand side of S contains at most one occurrence of every variable;
- LHS-disjoint, if the left-hand sides of inequalities are pairwise disjoint w. r. t. their variables.;
- LR-disjoint, if the left-hand side and the right-hand side of each inequality is disjoint w. r. t. their variables;
- simply ordered, if there exists a partial order $<$ on the variables occurring in S such that if x occurs in the left-hand side and y on the right-hand side of one and the same inequality then $x < y$;
- strongly ordered, if there exists an equivalence relation $\cong$ on all the variables in S and a partial order $<$ on the equivalence classes $[x], [y], \ldots$ of $\cong$ such that
  - if x and y occur on the right-hand side of one and the same inequality then $x \cong y$;
  - if x occurs on the left-hand side and y on the right-hand side of one and the same inequality then $[x] < [y]$.

While it may seem at first that disjointness of variables either between the left-hand sides and the corresponding right-hand sides of inequalities (LR-disjointness) or between different left-hand sides (LHS-disjointness) should result in significantly simplifying semi-unification, it is easily shown that this is not so; similarly, for right-linear and simply ordered systems of equalities.

Theorem 5 *General semi-unification is (log-space) reducible to each of right-linear, LHS-disjoint, LR-disjoint, and simply ordered semi-unification.*

Strongly ordered systems of inequalities on the other hand make it possible to solve one inequality at a time.

Theorem 6 *Strongly ordered semi-unification is decidable in time $O(n^k)$ where n is the total size (e.g., number of symbols) of the input and k is the number of inequalities in the input.*

3.3 Type Inference for Programs with Small Types

The Damas-Milner Calculus is provably intractable[11] [KM89], which is in contrast to the overall positive practical experience with languages based on ML's type rules. We shall attempt to argue that the apparent practicality of polymorphic type inference in the face of theoretical infeasibility results is not coincidental.

A conventional remedy for eliminating problems with type inference is to mandate explicit, fully typed declarations of variables, parameters and other basic syntactic units. Observe, for example, that type checking in the "explicit" Second Order λ-calculus is easy in the sense that there is a fast polynomial time algorithm for checking the type correctness of a fully typed λ-expression. Applying this sort of remedy to the Milner-Mycroft Calculus highlights, though, why type *checking* (with explicit type information embedded in the program) is, in general, no more "practical" from a user's point of view than type *inference* (with no or only optional type information in the program). After all, writing a 200-line (untyped) program whose principal typing is bigger (measured, for example, in terms of the "dag size" of the principal type) than the number of atoms in the universe can hardly be considered more impractical than writing the program *with* this typing information in the first place! Even though both these cases seem to have the same "intuitive" complexity they are treated very differently in conventional complexity analysis since the two input sizes are dramatically different.

The formalization of type inference in logical calculi does not take the *intensional* character of types and typings into account. *Types* and typings are generally viewed as abstractions of the *behavior* of programs and their parts, and — by analogy to types and program behaviors — *type descriptions* are meant to be abstractions of the *(syntactic) programs* themselves. If the complete inferred type information of a program is, in general, exponentially bigger than the (untyped) program itself, we think it unreasonable to say the type information is an *abstract* description of the program. Either the type description mechanism is inadequate for capturing the intended abstraction of behavior or the program at hand does *not* have a suitable abstract description of its behavior. The first explanation points toward a problem with the whole language, an issue that will have to be addressed by language designers. Given a fixed static typing discipline, however, and its implicit insistence that only behavior that is expressible in it should be considered desirable, the second explanation can be interpreted as saying that

[11] Intractable is used here in the sense of "being hard for (at least) NP or co-NP"

the program at hand has no "reasonable" abstract description of its behavior and thus should be considered unacceptable — type-incorrect. If we require that a λ-expression e only be considered "effectively well-typed" whenever it is typable in the sense of the Milner-Mycroft Calculus *and* its complete (principal) type information is at most polynomially bigger than e itself, then we can show that effective well-typing is (theoretically) feasible. The rationale behind this decision could be formulated, somewhat provocatively, as "(good) programs have small types". A similar argument has been suggested by Boehm [Boe89], and the observations about ML programs made in [KM89] are consistent with our explanation.

Consider the type inference system in Table 3 in Appendix A, which we shall call the *explicit* Milner-Mycroft Calculus. The only difference from the Milner-Mycroft Calculus is that all binding occurrences of variables x must have an explicit associated type embedded in the λ-expression.

We can define notions of typability and type inference as usual. Typed λ-expressions are defined by the grammar

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

where τ ranges over monotypes, and σ over polytypes. For every typed λ-expression e there is a unique underlying untyped λ-expression, $\bar{e}$, derived by erasing all mention of types in the typed λ-expression (and all colons); e is called a *typed version* of $\bar{e}$. Clearly, every typed λ-expression has a principal type in the explicit Milner-Mycroft Calculus with respect to a given type assignment. We have the following proposition.

Proposition 2 *There is a polynomial time algorithm for computing the principal type of a typed λ-expression or indicating untypability.*

We can now formally define a size-bounded restriction of the Mycroft Calculus. Let p be a fixed polynomial of one variable, and let $|e|$ be the number of symbols in a typed or untyped λ-expression e, and let eMM stand for the explicit Mycroft Calculus. Define

$$MM^p = \{\bar{e} : \exists A, \sigma \mid eMM \vdash A \supset e : \sigma \text{ and } |e| \leq p(|\bar{e}|)\}$$

A simple way to think about this set is to recognize that, if $A \supset e : \sigma$ is derivable in eMM, then $A \supset \bar{e} : \sigma$ is derivable in MM. The second

requirement encodes the fact that MM^p considers only those untyped λ-expressions type-correct that have a typed version whose type information is at most polynomially bigger than the untyped λ-expression itself. This is not to suggest that such a global property is a good *definitional requirement* on type systems – certainly not – but that the size bound is a good *property* of a type system for programs. It remains to be seen whether such a type system can be defined in a syntax-directed fashion.

Even though typability in the Milner-Mycroft Calculus is known to be intractable, possibly even undecidable, its size-bounded restriction MM^p is tractable.

Theorem 7 *MM^p is polynomial-time decidable.*

This theorem is actually stronger than stated. The upper bound on the running time is essentially determined by the polynomial p; that is, for given p there is an algorithm that decides MM^p in time $O(p)$. It would be interesting to see whether this theorem extends to the case where (monomorphic/polymorphic) type abbreviations of the form **let type** $s = \tau$ **in** ... are allowed, which are erased when forming $\bar{e}$ from an expression e.

If we consider the "typing" problem[12] of determining whether there is an assignment of (polynomial-sized) type expressions to function definitions in a language with Ada-style overloading, but without explicit type declarations (Ada requires such explicit declarations), it can be shown that this problem is NP-complete [ASU86, exercise 6.25], whereas the size-bounded polymorphic type inference problem is in P. This lends technical expression to the intuition that "overload resolution" as above is much harder than polymorphic type inference.

A Type Inference Systems

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

e ::= x | $\lambda\ x.e$ | (ee') |
let $x = e'$ **in** e |
fix $x.e$

[12]Some people would not consider this overload resolution problem an example of a typing problem.

The expression $\mathbf{fix}\, x.e$ essentially corresponds to a recursive definition of x with body e[13] The type expressions are formed according to the following productions.

$$\begin{aligned} \tau &::= \alpha \mid \tau \to \tau \\ \sigma &::= \tau \mid \forall \alpha.\sigma \end{aligned}$$

where α ranges over an infinite set TV of *type variables*, and $\forall$ is a (type) variable binding operator. The type expressions M derivable from τ are the *monotypes*;[14] the type expressions Π derivable from σ are called *polytypes*. A *type assignment* (or *type environment*) A is a mapping from a finite subset of V (variables) to Π (polytypes). For given A we define

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

A *typing* consists of three parts: a type assignment A, an expression e, and a type expression σ, written as $A \supset e : \sigma$. The Curry-Hindley Calculus, Damas-Milner Calculus, and Milner-Mycroft Calculus are type inference system given axiomatically in tables 1 and 2.

B Proof Sketch

This is a brief sketch of the proof of theorem 4. It is based on showing that the general-purpose semi-unification algorithm presented in [Hen89, chapter 6] can be "sped up" for left-linear semi-unification. A direct proof, not involving the general algorithm, is also possible, but is less intuitive in our view. In [Hen88] we presented a graph-theoretic algorithm that computes most general semi-unifiers of arbitrary systems of (equations and) inequalities. Terms are represented by term graphs, equations by equivalence relations on the nodes and inequalities by colored "arrows" — additional directed edges — in the term graph. Every color corresponds to a different inequality. See Figure 1 to get a general idea about the nature of these closure rules.

For *left-linear* systems, it can be shown by induction on arrow graph rewriting steps that the algorithm *never* executes any one of the two rules

[13] This only defines the value, but does not yield a binding to x.

[14] Note that, in contrast to [Mil78] and [Myc84] our monotypes can contain (necessarily free) occurrences of type variables.

Let A range over type environments; x over variables; e, e' over λ-expressions; α over type variables; τ, τ' over monotypes; σ, σ' over polytypes. The following are type inference axiom and rule schemes.

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

Table 1: Type inference axioms and rules

Axiom/rule	Cur-Hin	Dam-Mil	Mil-Myc
TAUT	$\surd$	$\surd$	$\surd$
GEN	$\surd$	$\surd$	$\surd$
INST	$\surd$	$\surd$	$\surd$
ABS	$\surd$	$\surd$	$\surd$
APPL	$\surd$	$\surd$	$\surd$
LET-M	$\surd$		
LET-P		$\surd$	$\surd$
FIX-M	$\surd$	$\surd$	
FIX-P			$\surd$

The mark $\surd$ indicates the corresponding axiom/rule is present in the calculus in whose column it appears; blank space means it is not included. The Flat Mycroft Calculus is restricted to λ-expressions with no **let**-operator and with only one occurrence of a **fix**-operator, which must occur at top-level.

Table 2: The Curry-Hindley, Damas-Milner, Milner-Mycroft type inference calculi

that merge equivalence classes. In other words, no merging takes place at all. Since the color information is only needed to trigger such merges correctly, it is superfluous! Consequently, we can compute the transitive closure of the initial arrows and apply rule 2, which propagates arrows "downwards" to the children. After each application of rule 2 we recompute the transitive closure and continue until no more applications of rule 2 are possible. At this point, we check if the computed (transitive) arrow structure together with the original term graph is acyclic. If the ranked alphabet is has only one functor, we are done at this point: If the arrow graph is is acyclic or all cycles contain only arrows, but no term graph edges, then the original left-linear system of inequalities has a semi-unifier; otherwise it doesn't. If the ranked alphabet is more complicated and allows for possible functor clashes we check after acyclicity testing whether there is a functor clash. This can be done by a slightly modified unification algorithm without "occurs" check. Clearly, the computational bottleneck is the dynamic transitive closure computation. Using any one of well-known dynamic transitive closure algorithms [**?**, Yel88], a naive analysis yields an $O(n^3)$ running time where n is the number of symbols in the given semi-unification problem. It remains to be seen whether this bound can be improved by exploiting the

Let A range over type environments; x over variables; e, e' over λ-expressions; α over type variables; τ, τ' over monotypes; σ, σ' over polytypes. The following are type inference axiom and rule schemes.

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

Table 3: Type inference axioms and rules for explicit Milner-Mycroft Calculus

specific structure of the arrow graphs.

Figure 1: Closure rules

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.

[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.

[BY79] C. Ben-Yelles. *Type-assignment in the Lambda-calculus*. PhD thesis, University College, Swansea, 1979.

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

[CL73] C. Chang and R. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York and London, 1973.

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

[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, January 1982.

[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.

[GMM+78] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. In *Proc. 5th ACM POPL*, pages 119–130, 1978.

[Hen87] Fritz Henglein. The Milner-Mycroft Calculus

is tractable. Manuscript, December 1987.

[Hen88] Fritz Henglein. Type inference and semi-unification. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 184–197, New York, NY, USA, July 1988. ACM.

[Hen89] Fritz 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.

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

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

[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.

[KMNS88] D. Kapur, D. Musser, P. Narendran, and J. Stillman. Semi-unification. In *Proc. Foundations of Software Technology and Teoretical Computer Science*, pages 435–454. Springer, December 1988. Lecture Notes in Computer Science, Vol. 338.

[KTU88a] A. Kfoury, J. Tiuryn, and P. Urzyczyn. On the computational power of universally polymorphic recursion. In *Proc. Symp. on Logic in Computer Science*, pages 72–81, June 1988.

[KTU88b] 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, January 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.

[Lei83] D. Leivant. Polymorphic type inference. In *Proc. 10th ACM Symp. on Principles of Programming Languages*, pages 88–98. ACM, January 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, October 1987.

[Lei89] H. Leiß. Decidability of semi-unification in two variables. Technical Report INF-2-ASE-9-89, Siemens, Munich, Germany, July 1989.

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

[Mac88] D. MacQueen. Personal communication, June 1988.

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

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

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

[MP85] L. Meertens and S. Pemberton. Description of B. *SIGPLAN Notices*,

20(2):58–76, February 1985.

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

[PM88] D. Parker and R. Muntz. A theory of directed logic programs and streams. In *Proc. Int'l Conf. on Logic Programming*, pages 620–650, August 1988.

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

[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, December 1986.

[Wan84] M. Wand. A semantic prototyping system. *Proc. ACM SIGPLAN '84 Symp. on Compiler Construction, SIGPLAN Notices*, 19(6):213–221, June 1984.

[Yel88] D. Yellin. A dynamic transitive closure algorithm. Technical Report RC 13535, IBM T.J. Watson Research Ctr., June 1988.