Simple Type Inference and Unification*

Fritz Henglein†
Courant Institute of Mathematical Sciences
New York University
715 Broadway, 7th floor
New York, N.Y. 10012, USA
Internet: henglein@nyu.edu or henglein@paul.rutgers.edu

October 12, 1988

Abstract

Kanellakis and Mitchell [KM89] have shown that a form of extended unification can be encoded as a typing problem in the programming language ML. We flesh out in detail their construction for ordinary unification and simple type inference and show that unification is reducible to simple type inference by a one-way finite state transducer. This implies that lower bounds for unification extend to simple inference and, in particular, that simple type inference is log-space hard for P.

1 Introduction

It has long been known that simple type inference and unification are closely related. Morris [Mor68] showed that simple type inference — type inference in the (Simply) Typed λ-calculus [Hin69, Cur69] — can be efficiently reduced to solving a set of equations over a free term algebra (see also [Han87]). Apparently not aware of Robinson's work on unification [Rob65], Morris laid out an *ad hoc* algorithm for solving the derived equations. Hindley [Hin69] and Curry [Cur69] built on Robinson's unification lemma to prove that the Simply Typed λ-calculus admits most general typings for every typable expression, a characteristic that is generally known as the *Principal Typing* property. The exponential running

*New York University, SETL Newsletter 232, Oct. 1988

†This research has been supported by the ONR under contract number N00014-85-K-0413. Part of this work was done while the author was a summer visitor at IBM T. J. Watson Research Center.

time of Robinson's original unification algorithm (and Morris's *ad hoc* algorithm) was improved to almost linear [HK71] and finally to linear time [PW78, MM82] over a compact directed acyclic graph *(dag)* representation of first-order terms. It is apparently also generally known that the reduction from type inference to unification can be done efficiently, which usually means in linear time on a random access machine (RAM).

With the advent of parallel processing and a robust definition of problems amenable to fast parallel processing, namely the class NC [Coo80], the quest for a fast parallel unification algorithm culminated in a result that presented strong evidence that no such algorithm exists. More precisely, Dwork, Kanellakis, and Mitchell [DKM84] showed that unifiability is hard for P with respect to log-space reductions. Under the generally accepted assumption that $NC \neq P$ this implies that unifiability is not in NC. Since, with the exception of [KM89], only reductions from type inference to unification have been considered, this does not preclude the existence of a fast parallel algorithm for simple type inference.

In this paper we show that unification is efficiently reducible, in a very strong sense, to simple type inference. More specifically, we prove that unifiability is 1FSM-reducible [Yap86] (reducible by a finite state transducer with a one-way input head) to typability in the Simply Typed λ-calculus. As a corollary, this proves that simple type inference is hard for P with respect to log-space reductions. Note that a somewhat terse description of this reduction has already appeared in [KM89] where it was used to show that type checking for ML, even for its functional core only, is PSPACE-hard. We provide some technical details and present some refined results in this paper.

The reduction is rather straightforward, even though some care is required to prove the reduction correct and even though an attempt at a naive translation of terms into type expressions would be hopeless due to the Howard-Curry isomorphism of types and theorems in intuitionistic propositional calculus [How80].

The "real-time reducibility" of unification to simple type inference can be viewed as expressing that unification is a subproblem of simple type inference, but not vice versa.[1]

The outline of this paper is as follows. In section 2 we quickly review unification, in section 3 simple type inference. We present a quick description of the standard problem representations in section 4, and show, in section 5, a preliminary simplification of the unification problem. Section 6 contains the reduction from the simplified unification problem to simple type inference. In section 7 we briefly address the reducibility of simple type inference to unification and compact directed acyclic graph repre-

[1] This is an attempt at putting the reducibility results into an intuitive form; it is not to be construed as a formal definition of "subproblem".

sentations. Finally, section 8 presents some conclusions and an outlook on other close connections between type checking and unification-style problems.

2 Unification

Unification is the problem of solving equations between first-order terms with variables. For a thorough investigation of the elementary theory of unification, see [LMM87]. We only present the definitions of importance to us here.

Let F, C, and V be three pairwise disjoint denumberable infinite sets, called *functors*, *constants*, and *variables*, respectively. The set of *(first-order) terms* $T(F, C, V)$ (or simply T whenever F, C, and V are understood) consists of all strings derivable from M in

$$M ::= x|c|f(M, \ldots, M)$$

where f, c, and x range over F, C, and V, respectively.

Two terms M_1, M_2 are *equal*, denoted $M_1 = M_2$, if and only if M_1 and M_2 are identical as strings. A *substitution* is a mapping from V to T that is the identity almost everywhere. Every substitution σ can be applied to terms by recursively defining

$$\begin{aligned} \sigma(c) &= c, \text{ if } c \in C \\ \sigma(f(M_1, \ldots, M_n)) &= f(\sigma(M_1), \ldots, \sigma(M_n)). \end{aligned}$$

Two terms M_1, M_2 are *unifiable* if there is a substitution σ, called *unifier*, such that $\sigma(M_1) = \sigma(M_2)$. The *unifiability problem* is the problem of deciding whether or not two input terms are unifiable.

If $M_1 = g(x, g(x, y))$, $M_2 = g(f(y), z)$, and $M_3 = g(f(x), c)$, then M_1 and M_2 are unifiable, but M_1 and M_3 are not.

3 Simple Type Inference

Simple type inference is the problem of checking for type correctness in a monomorphic language that admits elision of type declarations for program variables. This problem is formalized in [CF58], [Mor68], [Hin69], and [Cur69], and is treated from a (functional) programming language point of view in [Han87]. We only present the definitions necessary for our problem.

A *λ-expression* is any string derivable from e in

$$e ::= x|\lambda x.e|ee|(e)$$

where x ranges over a denumberable set I of *program variables*. A λ-expression of the form $e_1e_2e_3$ associates to the left; i.e., it is equivalent to $(e_1e_2)e_3$. The free program variables in λ-expression e are denoted by $FV(e)$. A *type expression* is any string derivable from τ in

$$\tau ::= t|\tau \to \tau|(\tau)$$

where t ranges over a denumerable set T of *type variables*. Whereas juxtaposition in λ-expressions associates to the left, the $\to$ type constructor associates to the right. A *type environment (type assumption)* is a mapping from a finite subset of I to type expressions. If A is a type environment and Θ is a subset of I then $A|_\Theta$ is the restriction of A to the domain Θ and $A\{x : \tau\}$ denotes the type environment defined by

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

There are also substitutions (on type variables) that map type variables to type expressions and, by extension, type expressions to type expressions and type environments to type environments.

A *typing* is a triple, written $A \supset e : \tau$, where A is a type environment, e a λ-expression, and τ a type expression. Typings are the logical statements in a type inference system that syntactically characterizes type correctness. In this sense $A \supset e : \tau$ should be pronounced "Expression e has type τ in type environment A". The type inference rules of the Simply Typed λ-calculus are presented below. Here we assume that all parentheses in λ-expressions and type expressions are present to make associativity explicit.

(TAUT) $A\{x : \tau\} \supset x : \tau$

(APPL)
$$\frac{A \supset e_1 : \tau_1 \to \tau_2 \quad A \supset e_2 : \tau_1}{A \supset (e_1e_2) : \tau_2}$$

(ABS)
$$\frac{A\{x : \tau_1\} \supset e : \tau_2}{A \supset \lambda x.e : \tau_1 \to \tau_2}$$

A typing is *valid* if it is derivable by application of instances of the axiom schema TAUT and the two rule schemas APPL and ABS. A λ-expression e is *(simply) typable* if and only if there exists a type environment A and a type expression τ such that $A \supset e : \tau$ is valid. The

typability problem is the problem of deciding whether a given λ-expression is typable.

Theorem 1 *(Principal Typing Property)*
For every simply typable λ-expression e there are A_e and τ_e such that

- *$A_e \supset e : \tau_e$ is valid, and*
- *for every valid typing $A \supset e : \tau$ there exists a substitution σ (on type variables) such that $\sigma(A_e) = A|_{FV(e)}$ and $\sigma(\tau_e) = \tau$.*

The typing $A_e \supset e : \tau_e$, if it exists, is called *principal* for e.

Proof: See [Hin69].

If $e_1 = \lambda x.\lambda y.(xy)$ and $e_2 = \lambda x.(xx)$, then e_1 is typable, but e_2 is not. With $A_{e_1} = \{\}$, the everywhere-undefined mapping, and $\tau_{e_1} = (t_1 \to t_2) \to (t_1 \to t_2)$, a principal typing for e_1 is $A_{e_1} \supset e_1 : \tau_{e_1}$.

4 Standard Representations

Representations of problems are usually left implicit or considered part of the problem itself. This is acceptable in those cases where different representations can be transformed into each other in, say, logarithmic space and the complexity classes considered are closed with respect to such transformations [Yap86]. Since we will be dealing with extremely resource-bounded computations — finite state machines — the representations of terms and λ-expressions have to be carefully defined. We present here what we consider the standard string representations. The more compact, "preprocessed" directed-acyclic-graph representations are addressed in section 7.

4.1 Standard Representation of Terms

To describe instances of the unifiability problem, let Σ_I = { '(', ')', ',', 'f', 'x', '0', '1', '=' } be the input alphabet. Without loss of generality we assume that the set of functors F is $\{f_1, \ldots, f_i, \ldots\}$ and the set of variables V is $\{x_1, \ldots, x_i, \ldots\}$. A functor f_i is then represented by the string "f $\bar{i}$" $\in \Sigma_I^*$ where $\bar{i}$ is the binary representation of i. Similarly the representation of a variable x_i is "x $\bar{i}$" $\in \Sigma_I^*$. Terms are represented in a straightforward fashion. The representation of $f_5(f_2(x_1), x_2)$ is "f101(f10(x1),x10)" $\in \Sigma_I^*$. An instance of the unifiability problem consists of the representations of two terms separated by the character '='; for example, "f101(f10(x1),x10)=f101(x10,x11)" $\in \Sigma_I^*$.

4.2 Standard Representation of λ-expressions

To describe instances of the simple typability problem, viz. λ-expressions, let $\Sigma_O = \{$ '(', ')', 'λ', '.', 'f', 'g', 'x', '0', '1' $\}$ be the output alphabet. We assume that the set of program variables is $\{f, g, x_1, \ldots, x_i, \ldots\}$[2]. The representation of program variables is the same as for variables in terms. The λ-expression $\lambda f.\lambda x_6.(x_4 x_6)$ is represented by the string "λf.λx110.(x100x110)" $\in \Sigma_O^*$.

5 Simplification of the Unifiability Problem

We will reduce the unifiability problem in little steps of 1FSM-reductions. Since 1FSM-reductions are *proper* [Yap86], their composition is also a 1FSM-reduction. In this section we show that unification problems can be "normalized" by restricting F and C to a single functor $[\ldots]$ and two constants $0, 1$.

5.1 Elimination of Functors

Let $T = T(F, C, V)$ as before and let $T' = T(\{[\ldots]\}, F \cup C, V)$ where $[\ldots]$ is a new functor not already in F. Define the transformation function $\mu_1 : T \to T'$,

$$\begin{array}{lcl} \mu_1(x) & = & x, \text{ if } x \in V \\ \mu_1(c) & = & c, \text{ if } c \in C \\ \mu_1(f(M_1, \ldots, M_n)) & = & [f, [\mu_1(M_1), \ldots, \mu_1(M_n)]], \text{ otherwise} \end{array}$$

We have the following lemma.

Lemma 1 *For all $M_1, M_2 \in T$, M_1 and M_2 are unifiable if and only if $\mu_1(M_1), \mu_1(M_2) \in T'$ are unifiable.*

Proof: By structural induction on T.

The translation of $f_5(f_2(x_1), x_2)$ via μ_1 returns $[f_5, [[f_2, [x_1]], x_2]]$. It is easy to see that μ_1 can be implemented by a one-way finite state transducer (1FSM-reduction).

5.2 Elimination of Constants

Without loss of generality we can assume now that $F \cup C$, the set of constants in $T(\{[\ldots]\}, F \cup C, V)$, is represented by the binary numerals $\{0, 1\}^*$. The numerals can also be eliminated by encoding them as lists

[2]Note that the program variables comprise V, the variables, and the extra elements f and g.

— with the constructor $[\ldots]$ — over the binary alphabet $\{0,1\}$. Let $T'' = T(\{[\ldots]\}, \{0,1\}, V)$.

$$\begin{array}{lcll} \mu_2(x) & = & x, & x \in V \\ \mu_2(c) & = & [b_1, \ldots, b_k], & c = b_1 \ldots b_k \in \{0,1\}^* \\ \mu_2([M_1, \ldots, M_n]) & = & [\mu_2(M_1), \ldots, \mu_2(M_n)] & \end{array}$$

The correctness of this transformation is guaranteed by the next lemma.

Lemma 2 *For all $M_1, M_2 \in T'$, M_1 and M_2 are unifiable if and only if $\mu_2(M_1), \mu_2(M_2) \in T''$ are unifiable.*

Proof: By structural induction on T'.

The encoding of $[c_5, [[c_2, [x_1]], x_2]]$ via μ_2 is $[[1,0,1], [[[1,0], [x_1]], x_2]]$. Again, this translation can be implemented by a one-way finite state machine.

6 Reduction of Unification to Simple Type Inference

The basic idea behind the reduction of unifiability to simple typability is to establish a correspondence of terms with type expressions that are the principal types of some λ-expressions, and to encode the equality of terms with the typing rules of the Simply Typed λ-calculus. Some care is necessary to make this idea work since the Howard-Curry isomorphism of types and theorems in intuitionistic propositional calculus [How80] shows that not every type expression is a principal (or nonprincipal) type of some λ-expression. For example, consider the term $f(f(x,x),x)$. We might feel tempted to try to construct a λ-expression with principal type $(x \to x) \to x$. But no such λ-expression exists since $(x \to x) \to x$ is not a theorem in the intuitionistic propositional calculus, not even in the classical propositional calculus. Instead, we will use a standard construction for encoding lists in the λ-calculus that has principal type $(((x \to x \to \alpha_1) \to \alpha_1) \to x \to \alpha_2) \to \alpha_2$. After the simplifications above we can assume that instances of the unifiability problem are drawn from terms in $T'' = T(\{[\ldots]\}, \{0,1\}, V)$.

6.1 The Encoding of Terms by λ-expressions

We shall now define an encoding ρ of terms by λ-exressions.

$$\begin{array}{lcl}\rho(x) & = & x, \text{ if } x \in V \\ \rho(0) & = & \lambda x.\lambda y.x \\ \rho(1) & = & \lambda x.\lambda y.\lambda z.x \\ \rho([M_1, \ldots, M_n]) & = & \lambda f.(f\rho(M_1)\ldots\rho(M_n))\end{array}$$

For example, the encoding of term $[[1,0,1],[[[1,0],[x_1]],x_2]]$ via ρ yields

$$\begin{array}{ll}\lambda f.(f & \lambda f.(f\lambda x.\lambda y.\lambda z.x\lambda x.\lambda y.x\lambda x.\lambda y.\lambda z.x) \\ & \lambda f.(f\lambda f.(f\lambda f.(f\lambda x.\lambda y.\lambda z.x\lambda x.\lambda y.x)\lambda f.(fx_1))x_2))\end{array}$$

6.2 The Encoding of Equality

Now that we know how to represent terms it remains to show how the equational constraint in a unification problem can be captured by a λ-expression. This is where the typing rules of the Simply Typed λ-calculus come in. Let us first note that the structure of a λ-encoding $\rho(M)$ is directly reflected in the valid typings for $\rho(M)$ since $\rho(M)$ contains only λ-abstractions, no applications. For example, the term $[[1],[x_1]]$, originally representing $f_1(x_1)$, would give rise to the principal typing

$$\begin{array}{l}\{x_1 : t_1\} \supset \lambda f.(f\lambda f.(\lambda x.\lambda y.\lambda z.x)\lambda f.(fx_1)) : \\ \quad ((u_1 \to u_2 \to u_3 \to u_1) \to ((t_1 \to u) \to u) \to u') \to u'\end{array}$$

The typing rules of the Simply Typed λ-calculus stipulate that all occurrences of a λ-bound variable have the same type in any typing derivation. This constraint is exploited to characterize the equational constraint in the underlying unification problem. More precisely, we extend ρ to map unifiability problem instances to λ-expressions as follows.

$$\rho(M_1 = M_2) \quad = \quad \lambda g.\lambda f.(f(g\rho(M_1))(g\rho(M_2)))$$

For example, $f_1(x_1) = x_2$ is mapped to

$$\lambda g.\lambda f.(f(g\lambda f.(f\lambda f.(\lambda x.\lambda y.\lambda z.x)\lambda f.(fx_1)))(gx_2)).$$

6.3 Correctness

It is easy to see that ρ can be computed by a one-way finite state transducer. To complete the reduction from unifiability to simple typability, it remains to be shown that ρ is indeed a problem reduction; that is, for all $M_1, M_2 \in T(\{[\ldots]\}, \{0,1\}, V)$ it holds that M_1, M_2 are unifiable if

and only if there is a valid typing for $\rho(M_1 = M_2)$ in the Simply Typed λ-calculus.

The intuition is quite simple: the definitions of $\rho(0), \rho(1)$ and $\rho([M_1, \ldots, M_k])$ guarantee that no two types of these representations are unifiable. Consequently, we expect that if the types of two λ-encodings indeed unify then this unifier can be translated back to a unifier of the underlying terms. This is indeed the case, but the formalization of this idea is more cumbersome than enlightening.

There are many possible proofs of correctness. For example, we can try to show that the principal typings of $\rho(M_1)$ and $\rho(M_2)$ are unifiable if and only if M_1, M_2 are unifiable. This is technically rather messy since there are in general many more variables in the principal typings than in the underlying terms. We take a slightly different route.

6.3.1 Unifiability Implies Typability

First we show that if a pair of terms is unifiable then the λ-representation of this unifiability problem is simply typable.

Define the canonical type mapping τ that maps type environments and terms to type expressions as follows.

$$\begin{array}{lcl} \tau(A, x) & = & A(x), x \in V \\ \tau(A, 0) & = & \alpha \to \beta \to \alpha \\ \tau(A, 1) & = & \alpha \to \beta \to \gamma \to \alpha \\ \tau(A, [M_1, \ldots, M_k]) & = & (\tau(A, M_1) \to \ldots \to \tau(A, M_k) \to \alpha) \to \alpha \end{array}$$

Here α, β, γ denote fixed type variables. The following proposition is easy to prove by structural induction over terms.

Proposition 3 *Let A, A' be type environments, and $M, N_1, \ldots, N_k$ terms whose variables are contained in the domain of A.*

1. *$\tau(A, M)$ is well-defined and unique.*
2. *If A is injective then τ is injective with respect to its second argument; i.e., $\tau(A, N_1) = \tau(A, N_2)$ implies $N_1 = N_2$.*
3. *The typing $A \supset \rho(M) : \tau(A, M)$ is valid.*
4. *If $\{x_1, \ldots, x_n\}$ is the domain of A then $A = \{x_1 : \tau(A, x_1), \ldots, x_n : \tau(A, x_n)\}$*

Given a substitution σ on *terms* (not type expressions) we define $\sigma(A)$, the application of σ to a type environment $A = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, as follows.

$$\sigma(A) = \{x_1 : \tau(A, \sigma(x_1)), \ldots, x_n : \tau(A, \sigma(x_n))\}.$$

Note that according to proposition 3, part 4, $\iota(A) = A$ for all A where ι denotes the identity ("empty") substitution.

Lemma 4 *For all terms M, type environments A, and term substitutions σ,*

$$\tau(A, \sigma(M)) = \tau(\sigma(A), M)$$

or both $\tau(A, \sigma(M))$ and $\tau(\sigma(A), M)$ are undefined.

Proof: We prove this lemma by structural induction on M, assuming $\tau(A, \sigma(M))$ and $\tau(\sigma(A), M)$ are defined.

- (Base cases) If M is a variable, x_i, then

$$\begin{aligned} \tau(\sigma(A), M) &= \tau(\sigma(A), x_i) \\ &= \sigma(A)(x_i) \\ &= \tau(A, \sigma(x_i)) \text{ (by definition of } \sigma(A)) \\ &= \tau(A, \sigma(M)) \end{aligned}$$

  The cases for $M = 0$ and $M = 1$ are trivial.

- (Inductive case) If $M = [N_1, \ldots, N_k]$ for some terms $N_1, \ldots, N_k$, then

$$\begin{aligned} \tau(\sigma(A), M) &= \tau(\sigma(A), [N_1, \ldots, N_k]) \\ &= (\tau(\sigma(A), N_1) \to \ldots \to \tau(\sigma(A), N_k) \to \alpha) \to \alpha \\ &= (\tau(A, \sigma(N_1)) \to \ldots \to \tau(A, \sigma(N_k)) \to \alpha) \to \alpha \text{ (ind. hyp.)} \\ &= \tau(A, [\sigma(N_1), \ldots, \sigma(N_k)]) \\ &= \tau(A, \sigma([N_1, \ldots, N_k])) \\ &= \tau(A, \sigma(M)) \end{aligned}$$

This completes the proof.

Lemma 5 *For all $M_1, M_2 \in T''$, if M_1 and M_2 are unifiable then $\rho(M_1 = M_2)$ is simply typable.*

Proof: By assumption of the lemma there is a unifier υ of M_1, M_2; i.e., $\upsilon(M_1) = \upsilon(M_2)$. Let A be a type environment whose domain contains sufficiently many variables. By proposition 3, part 3, both $\upsilon(A) \supset \rho(M_1) : \tau(\upsilon(A), M_1)$

and $\upsilon(A) \supset \rho(M_2) : \tau(\upsilon(A), M_2)$ are valid typings. According to lemma 4 and by the fact that υ is a unifier we have $\tau(\upsilon(A), M_1) = \tau(A, \upsilon(M_1)) = \tau(A, \upsilon(M_2)) = \tau(\upsilon(A), M_2)$. Call this type τ'. Consequently,

$$A'\{g : \tau' \to \alpha'\} \supset \lambda f.(f(g\rho(M_1))(g\rho(M_2))) : (\alpha' \to \alpha' \to \alpha) \to \alpha$$

and

$$A' \supset \rho(M_1 = M_2) : (\tau' \to \alpha') \to (\alpha' \to \alpha' \to \alpha) \to \alpha$$

are valid typings, the latter of which shows that $\rho(M_1 = M_2)$ is simply typable.

6.3.2 Typability Implies Unifiability

We now proceed to prove that if $\rho(M_1 = M_2)$, for given terms M_1 and M_2, is typable then M_1 and M_2 are unifiable.

Some preliminary results on the normalization of typings are helpful in facilitating a translation of types to terms and from typings to substitutions. The *normalization* function ν on types is defined as follows.

$$\nu(\tau) = \begin{cases} \tau, & \tau \text{ is a type variable} \\ \alpha \to \beta \to \alpha, & \tau = \tau_1 \to \tau_2 \to \tau_1 \text{ for some } \tau_1, \tau_2 \\ \alpha \to \beta \to \gamma \to \alpha, & \tau = \tau_1 \to \tau_2 \to \tau_3 \to \tau_1 \text{ for some } \tau_1, \tau_2, \tau_3 \\ (\nu(\tau_1) \to \ldots \to \nu(\tau_n) \to \alpha) \to \alpha, & \tau = (\tau_1 \to \ldots \to \tau_n \to \tau') \to \tau' \\ & \text{for some } \tau_1, \ldots, \tau_n, \tau' \\ \alpha \to \beta \to \alpha, & \text{otherwise} \end{cases}$$

Proposition 6 *1. ν is well-defined and unique.*

2. For any set of type expressions $\tau_1, \ldots, \tau_k$ there is an injective type environment A and terms $N_1, \ldots, N_k$ such that $\nu(\tau_i) = \tau(A, N_i)$ for all i such that $1 \leq i \leq k$.

The mapping ν can be extended to type enironments in the standard way: $\nu(A) = \{x_1 : \nu(\tau_1), \ldots, x_n : \nu(\tau_n)\}$ if $A = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$.

Lemma 7 *For any valid typing $A \supset \rho(M) : \tau$, the typing $\nu(A) \supset \rho(M) : \nu(\tau)$ is also valid, and $\nu(\tau) = \tau(\nu(A), M)$.*

Proof: This can be shown by simple induction on the structure of M.

Lemma 8 *For all $M_1, M_2 \in T''$, if $\rho(M_1 = M_2)$ is typable in the Simply Typed λ-calculus then M_1 and M_2 are unifiable.*

Proof: By assumption, there is a valid typing $A \supset \rho(M_1 = M_2) : \tau$ for $\rho(M_1 = M_2)$. By the definition of ρ this expands to

$$A \supset \lambda g.\lambda f.(f(g\rho(M_1))(g\rho(M_2))) : \tau.$$

Since the typing rules of the Simply Typed λ-calculus are syntax-directed, we can conclude, by "backwards reasoning", that there are type expressions τ', τ_2, τ_3 such that $\tau = (\tau' \to \tau_2) \to (\tau_2 \to \tau_2 \to \tau_3) \to \tau_3$ and, with $A' = A\{g : \tau' \to \tau_2, f : \tau_2 \to \tau_2 \to \tau_3\}$, both

$$A' \supset \rho(M_1) : \tau'$$

and

$$A' \supset \rho(M_2) : \tau'$$

are valid. Let us define $A'' = \nu(A')$ and $\tau'' = \nu(\tau')$. By lemma 7, the typings

$$A'' \supset \rho(M_1) : \tau''$$

and

$$A'' \supset \rho(M_2) : \tau''$$

are both valid. If $A'' = \{x_1 : \tau_1'', \ldots, x_k : \tau_k''\}$, proposition 6, part 2, implies that there are terms $M, N_1, \ldots, N_k$ and an injective type environment A_0 such that $\tau(A_0, M) = \tau'', \tau(A_0, N_1) = \tau_1'', \ldots, \tau(A_0, N_k) = \tau_k''$. If we define $\sigma = \{x_1 \to N_1, \ldots, x_k \to N_k\}$, the previous two typings can be rephrased as

$$\sigma(A_0) \supset M_1 : \tau(\sigma(A_0), M)$$

and

$$\{x_1 : \tau(A_0, \sigma(x_1)), \ldots, x_k : \tau(A_0, \sigma(x_k))\} \supset \rho(M_2) : \tau(A_0, M)$$

Also by lemma 7 we can conclude $\tau(\sigma(A_0), M_1) = \tau(A_0, M) = \tau(\sigma(A_0), M_2)$. Finally, this yields $\tau(A_0, \sigma(M_1)) = \tau(A_0, \sigma(M_2))$ by lemma 4 and, since A_0 is injective, by proposition 3, part 2, $\sigma(M_1) = \sigma(M_2)$. Consequently, M_1 and M_2 are unifiable.

Theorem 2 *For all $M_1, M_2 \in T''$, M_1 and M_2 are unifiable if and only if $\rho(M_1 = M_2)$ is simply typable.*

Proof: Lemma 5 shows one direction, lemma 8 the other.

7 Reductions and Directed Acyclic Graph Representations

Dwork, Mitchell, and Kanellakis [DKM84] proved that unifiability is hard for P with respect to log-space reductions. Their problem, however, assumes that the input is presented in the form of directed acyclic (labelled) dags. Similarly, λ-expressions can be represented by their abstract syntax trees with all occurrences of a program identifier merged with their corresponding binding. As an implication of [DKM84] and the above results we get the following theorem.

Theorem 3 *The problems of simple typability and unifiability are hard for P with respect to log-space reductions, for both the standard and the graph-based representations.*

Consequently all these problem/representation combinations are log-space interreducible, which shows that, in this sense, we are justified in speaking of "the" simple typability and unifiability problems irrespective of their specific respresentation.

Linear-time reductions are in practice of more importance than log-space reductions since speed is apparently higher valued than space. Since the log-space reduction in [DKM84] doesn't preserve space — i.e., it generates superlinear amount of output — it cannot possibly be executed in linear time, even on a (logarithmic-cost) RAM [AHU74]. Note that our reduction of unifiability to simple typability executes in linear time and no auxiliary space on any reasonable (sequential) computational model. Yet the linear-time reduction used to extract a unifiability problem from a λ-expression [Han87] uses more than logarithmic extra space. It is thus an open question whether there is a loglin-reduction from typability to unifiability; that is, a log-space reduction whose output size is $O(n)$ where n is the input size. Furthermore, the linear-time reduction is only linear time over the dag representation, since the standard representation of λ-expressions permits reuse of variable names. It can be shown that, in general, the bit-size of the output is $\Theta(n \log n)$ if the input bit-size is n and the representations are the standard ones.

It appears to be impossible for there to be a 1FSM-reduction from typability to unifiability in general, no matter what representation. Even if all variables in the input are disjoint and the expressions are fully parenthesized, it seems impossible to devise such a reduction since the recognition problem of a nonregular context-free language lurks in the background.

8 Conclusion and Outlook

It is commonly known that simple type inference is reducible to unification. We have shown that, conversely, unification is reducible, in a very strong sense, to simple type inference. Thus lower bounds for unification extend immediately to type inference. In particular, this implies that type inference is hard for P with respect to log-space reductions. The encodings we have used can be used to provide similar reductions of unification-like problems to typing problems. A case of this is the reduction of semi-unification [Hen88b] to type inference in the Simply Typed λ-calculus extended with a single additional, polymorphically typed fixed-point combinator [Hen88a]. This is a subproblem of type inference in the Milner-Mycroft Calculus, which permits arbitrary use of polymorphically typed fixed-point combinators and let-bindings and was shown to be reducible to semi-unification in [Hen88c]. The PSPACE lower bound of [KM89] on type inference in ML consequently extends to this subproblem of the Milner-Mycroft Calculus and, trivially, to typing in the Milner-Mycroft Calculus itself.

9 Acknowledgements

I wish express my thanks to Eric Allender for always lending an ear to my questions about computational complexity and for being such a treasure trove of knowledge and to Ken Perry for numerous discussions on unification and unification-like problems.

References

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[CF58] H. Curry and R. Feys. *Combinatory Logic*. Volume I of *Studies in Logic and the Foundations of Mathematics*, North-Holland, 1958.

[Coo80] S. Cook. Towards a complexity theory of synchronous parallel computation. In *Proc. Symposium ueber Logic und Algorithmik in honor of Ernst Specker*, Zuerich, Switzerland, February 1980.

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

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

[Han87] P. Hancock. Polymorphic type checking. In S. Peyton-Jones, editor, *The Implementation of Functional Programming Languages*, chapter 8, Prentice-Hall, 1987.

[Hen88a] F. Henglein. *Polymorphic Type Inference is Semi-Unification.* Technical Report (SETL Newsletter) 229, New York University, October 1988.

[Hen88b] F. Henglein. *Semi-Unification.* Technical Report (SETL Newsletter) 222, New York University, April 1988.

[Hen88c] F. Henglein. Type inference and semi-unification. In *Proc. ACM Conf. on LISP and Functional Programming*, ACM, ACM Press, July 1988.

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

[HK71] J. Hopcroft and R. Karp. *An Algorithm for Testing the Equivalence of Finite Automata.* Technical Report TR-71-114, Dept. of Computer Science, Cornell U., 1971.

[How80] W. Howard. The formulae-as-types notion of construction. In J. Seldin and J. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490, Academic Press, 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.

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

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

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

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

[Rob65] J. Robinson. A machine-oriented logic based on the resolution principle. *J. Assoc. Comput. Mach.*, 12(1):23–41, 1965.

[Yap86] C. Yap. Lecture notes on computational complexity theory. 1986. Unpublished manuscript.