Polymorphic Unification and ML Typing

Paris C. Kanellakis[1]
John C. Mitchell[2]
Technical Report No. CS-89-40
August 3, 1989

[1]Brown University, Dept. of Computer Science Providence, Rhode Island 02912
[2]Stanford University, Dept. of Computer Science Stanford, California 94305

Polymorphic unification and ML typing*

Paris C. Kanellakis†
Department of Computer Science
Brown University
Providence, RI 02912

John C. Mitchell‡
Department of Computer Science
Stanford University
Stanford, CA 94305

August 3, 1989

Abstract

We study the complexity of type inference for a core fragment of ML with lambda abstraction, function application, and the polymorphic `let` declaration. Our primary technical tool is the unification problem for a class of "polymorphic" type expressions. This form of unification, which we call *polymorphic unification*, allows us to separate a combinatorial aspect of type inference from the syntax of ML programs. After observing that ML typing is in DEXPTIME, we show that polymorphic unification is PSPACE hard. From this, we prove that recognizing the typable core ML programs is also PSPACE hard. Our lower bound stands in contrast to the common belief that typing ML programs is "efficient," and to practical experience which suggests that the algorithms commonly used for this task do not slow compilation substantially.

1 Introduction

A convenient feature of the programming language ML [GMW79,Mil85] is the way that type inference is used to eliminate the need for type declarations [Mil78]. When the programmer enters untyped code, the compiler responds with the type of the expression. For example, a programmer may declare the identity function by writing `let` $Id = \lambda x.x$. The complier then infers that Id has type $t \rightarrow t$, meaning that the identity maps any type t to itself. If the compiler cannot find a type for an expression, an error message is printed. Thus ML programmers receive the benefit of compile-time type checking (early detection of errors), without the inconvenience of supplying types explicitly. Since ML typing has proven useful in practice, the main ideas have also been adopted in other languages, such as Miranda [Tur85].

To simplify our analysis, we will focus on *core* ML expressions using only lambda abstraction, function application, and `let`. Since the main result is a lower bound, choosing a small fragment of ML makes our study more widely applicable. The ML type inference problem is: *given a core ML expression M, find a type for M if one exists, otherwise return* untypable. Our lower bound

---

*A preliminary version of this paper appeared in the *Proc. 16-th ACM Symp. on Principles of Programming Languages*, pages 105–115, January 1989.

†The work of this author was supported by: NSF grant IRI-8617344, ONR grant N00014-83-K-0146 ARPA Order No. 4786, and an Alfred P. Sloan Fellowship. Also, part of the work was completed, while this author was visiting the IBM T.J. Watson Research Center.

‡Supported by an NSF PYI Award.

will actually apply to the apparently simpler recognition problem: *given a core ML expression M, return* typable *if M has a type, else return* untypable. It is clear that any algorithm for the type inference problem also solves the recognition problem. A useful fact about ML typing is that when an expression M has a type, there is a *principal type* which indicates the form of all other types for M.

The main source of super-polynomial complexity is the `let` declaration, which is crucial to ML polymorphism. If we declare a function f by saying `let` $f = \ldots$, then different occurrences of f within this scope may be given different types. This is practically important, since it allows expressions such as

$$\texttt{let } f = \lambda x.x \texttt{ in } \ldots\ f(3)\ \ldots\ f(true)\ldots$$

in which a single function is applied to arguments of several types. Without `let`, ML type inference can be done efficiently. Using a linear time unification algorithm (as in [PW78], for example) we can compute the principle type of any `let`-free core ML expression in linear time. Even in this simple case, however, one must be careful with the representation of type expressions. To achieve linear time, types must be represented (and printed out) as directed acyclic graphs, or *dags,* since the string representation of a type may be exponentially longer than the given expression. Dag representations are a common data structure in unification [PW78,MM82,DKM84].

With `let`, we give a straightforward deterministic-time upper-bound that is exponential in the number of nested `let` declarations of the input. However, the type of an ML program may be doubly-exponential in its size, when written as a string, or singly-exponential when represented as a dag. This causes the usual ML type inference algorithms to have doubly-exponential worst-case behavior, since the type is usually printed out as a string. *A priori,* it might seem reasonable to look for a more succinct representation that would allow us to improve upon the exponential dag algorithm.

In studying the complexity of ML typing, we introduce a class of "polymorphic" type expressions that allow many types to be written concisely. We call the unification problem for these extended type expressions *polymorphic unification,* and show that determining the solvability of a polymorphic unification problem is PSPACE hard. By representing hard instances of polymorphic unification within ML programs, we then show that recognizing the typable core ML programs is PSPACE hard. It follows that no polynomial time algorithm can recognize the typable programs, unless P=PSPACE. This contradicts a quadratic time bound for the problem, claimed in [Lei83], and what appears to be a well-known "folk theorem," namely, that ML typing is linear time[1]. It also stands in contrast to the perceived efficiency of the algorithm in practice. The polymorphic unification proof uses an encoding of quantified propositional formulas, while the reduction to typing uses lambda calculus programming of approximately the same sophistication as the proof of Turing completeness for untyped lambda calculus [Bar84]. It remains to close the gap between our PSPACE lower bound and the exponential time upper bound.

The rest of this paper is organized as follows. Basic definitions of ML syntax, typing and unification are given in Section 2. Examples of programs with large types, and upper bounds, appear in Section 3. Extended type expressions and PSPACE hardness for polymorphic unification are presented in Section 4; PSPACE hardness for ML typing is discussed in Section 5. Concluding remarks appear in Section 6. For the reader who is not familiar with ML, typing rules and an algorithm for computing principal types are included in an Appendix.

---

[1]To the embarrassment of the second author, the incorrect "folk theorem" was put in print in [MH88].

2 ML expressions, types and unification

2.1 Core ML

The *core ML* expressions have the following abstract syntax

$$M \quad ::= \quad x \mid MM \mid \lambda x.M \mid \texttt{let}\ x = M\ \texttt{in}\ M,$$

where x may be any expression variable (*c.f.* [DM82,Mil78]). In writing expressions, we will adopt the usual conventions of lambda calculus. For example, MNP should be read as $((MN)P)$, and $\lambda x.MN$ read as $\lambda x.(MN)$.

In $\lambda x.M$ and $\texttt{let}\ x = N\ \texttt{in}\ M$, the variable x becomes bound in M. This leads to the renaming equivalences

$$(\alpha)_1 \qquad \lambda x.M \equiv \lambda y.[y/x]M,\ y \text{ not free in } M$$

$$(\alpha)_2 \qquad \texttt{let}\ x = N\ \texttt{in}\ M \equiv \texttt{let}\ y = N\ \texttt{in}\ [y/x]M,\ y \text{ not free in } M$$

where $[N/x]M$ denotes the result of substituting N for free occurrences of x in M (with renaming of bound variables to avoid capture, as usual). We say two expressions are *α-equivalent* if they differ only in the names of bound variables, and generally treat α-equivalent expressions as identical. An expression is *closed* if all variables are bound.

Reduction is a relation on α-equivalence classes of ML expressions which resembles symbolic execution. Reduction is axiomatized by

$$(\beta) \qquad (\lambda x.M)N \xrightarrow{\beta} [N/x]M$$

$$(let) \qquad \texttt{let}\ x = N\ \texttt{in}\ M \xrightarrow{let} [N/x]M$$

(There are also η-reduction rules, as in [Bar84], but they will not be needed in this paper.) Since $\texttt{let}\ x = M\ \texttt{in}\ N$ and $(\lambda x.N)M$ both reduce to $[M/x]N$, these expressions produce the same final value. However, there are different typing restrictions in ML.

We say M *let-reduces to* N, and write $M \xrightarrow{let} N$, if we can obtain N from M by repeatedly applying rule (*let*) to subexpressions, and renaming bound variables. If we can produce N from M using both (*let*) and (β), then we write $M \twoheadrightarrow N$. An interesting fact about let-reduction (only) is that it is finite Church-Rosser. The following proposition is essentially the uniqueness and finiteness of developments for untyped lambda calculus [Bar84], since every let in M may be regarded as a "marked" λ-redex. The reader is referred to [Bar84] for further discussion and proof.

Proposition 2.1 *Let M be any core ML expression. There is a unique* let*-free expression N such that every maximal sequence of* let*-reductions starting from M terminates at N. In particular, there are no infinite sequences of* let*-reductions.*

If N is a let-free expression obtained from M by repeated let-reduction, then we say N is a let *normal form* of M. By Prop 2.1, let normal forms are unique.

2.2 Types and typing assertions

The type expressions of core ML have the following form

$$\sigma \quad ::= \quad t \mid \sigma \to \sigma$$

where t may be any type variable. The standard syntactic convention is that $\to$ associates to the right. For example, $\sigma \to \tau \to \varrho$ should be read as $(\sigma \to (\tau \to \varrho))$.

The type of an expression depends on the types we assume for its free variables. For this reason, we use *typing assertions* of the form $\Gamma \triangleright M : \sigma$, where M is an ML program, σ is a type expression, and Γ is a type assignment, *i.e.*, a finite set

$$\Gamma = \{x_1 : \sigma_1, \ldots, x_k : \sigma_k\}$$

associating at most one type with each variable x. The assertion $\Gamma \triangleright M : \sigma$ may be read, "the expression M has type σ in context Γ."

We say M *is typable* if there is some *provable* typing assertion $\Gamma \triangleright M : \sigma$ about M. Typing assertions are proved using the ML *inference system*, which is summarized in the Appendix. Since the action of the ML type checker is more relevant to our lower-bound proof, the Appendix also includes an equivalent type inference algorithm.

The provable typing assertions are closed under substitution. For our purposes, a *substitution* will be a function from type variables to type expressions. A substitution S is applied to a type expression as usual, and to a type assignment Γ by applying S to every type expression in Γ. More specifically, $S\Gamma$ is the type assignment

$$S\Gamma = \{x : S\sigma \mid x : \sigma \in \Gamma\}.$$

A typing statement $\Gamma' \triangleright M : \sigma'$ *is an instance of* $\Gamma \triangleright M : \sigma$ if there exists a substitution S with

$$\Gamma' \supseteq S\Gamma, \text{ and } \sigma' = S\sigma.$$

Proposition 2.2 (Milner 78) *If $\Gamma' \triangleright M : \sigma'$ is an instance of a provable typing assertion $\Gamma \triangleright M : \sigma$, then $\Gamma' \triangleright M : \sigma'$ is provable.*

A typing assertion $\Gamma \triangleright M : \sigma$ is *principal typing for* M if it is provable, and has every provable typing assertion for M as an instance. When M is closed, the principal typing will have an empty type assignment, and so we say M has a *principal type*. It was first shown in [DM82] that every typable ML expression has a principal typing.

Proposition 2.3 (Damas–Milner 82) *If M is typable, then M has a principal typing.*

We will use the phrase *length of a typing* (or type expression) for the number of symbols we use to write the expression down on a piece of paper. It is sometimes useful to write $|\tau|$ for the length of type expression τ, and similarly for core ML expressions. Since a substitution cannot decrease the length of a typing, the principal typing of any expression has minimum length.

2.3 Unification and graph representation of type expressions

If E is a set of equations between type expressions, then a substitution S *unifies* E if $S\sigma = S\tau$ for every equation $\sigma = \tau \in E$. The unification algorithm of [Rob65] computes a most general unifying substitution, where S *is more general than* R if there is a substitution T with $R = T \circ S$.

Proposition 2.4 (Robinson 65) *Let E be any set of equations between type expressions. There is an algorithm* UNIFY *such that if E is unifiable, then* UNIFY(E) *computes a most general unifier. Furthermore, if E is not unifiable, then* UNIFY(E) *returns* failure.

An important part of the algorithm used to compute principal typings is the way that unification is used to combine typing statements about subexpressions. Specifically, if $M: \sigma \to \tau$ and $N: \varrho$, then we must unify σ and ϱ in order to type the application MN.

While most implementations of unification have slightly higher asymptotic running time, unification can be done in linear time [MM82,PW78]. To perform unification efficiently, it is common to represent the expressions to be unified (in our case, type expressions) as directed acyclic graphs (see [AHU83], for example). A directed acyclic graph (*dag*) representation is like the parse tree of an expression, with each node labeled by an operator or operand (in our case, $\to$ or a type variable). However, repeated subexpressions need be represented only once, resulting in nodes with indegree greater than one. It is easy to show that a dag of size n may represent an expression of length 2^n. Nonetheless, the time required for unification is *linear in the size of the dags* representing the expressions to be unified.

3 Lower bounds on type size and upper bounds on type inference

3.1 Lambda terms without "let"

Even without `let`, expressions may have principal types of exponential length. In constructing expressions with specific principal types, it is useful to adopt the abbreviation

$$\langle M_1, \ldots, M_k \rangle ::= \lambda z. z M_1 \ldots M_k$$

where z is a fresh variable not occurring in any of the M_i. This is a common encoding of sequences in untyped lambda calculus. It is easy to verify that if M_i has principal type σ_i, then the principal type of the sequence is

$$\langle M_1, \ldots, M_k \rangle : (\sigma_1 \to \ldots \to \sigma_k \to t) \to t$$

where t is a fresh type variable not occurring in any of the σ_i. We will write $\sigma_1 \times \ldots \times \sigma_k$ as an abbreviation for any $(\sigma_1 \to \ldots \to \sigma_k \to t) \to t$ with t not occurring in any σ_i.

Example 3.1 *The closed expression*

$$P ::= \lambda x.\ \langle x, x \rangle$$

has principal type $t \to (t \times t)$. If we apply P to an expression M with principal type σ, then the application PM will be typed by unifying t with σ, resulting in the typing $PM : \sigma \times \sigma$. Thus applying P to a typable expression doubles the length of its principal type.

By iterating the application of P from Example 3.1 to any typable expression, we can prove the following proposition.

Proposition 3.2 *For arbitrarily large n, there exist* `let`*-free closed expressions M of length n whose principal types have length $2^{\Omega(n)}$.*

Since the type of an expression may have many repeated subexpressions, the minimum-size dag representation of the type need not be exponential. In fact, using dag representations of types, we can compute principal typings in linear time. It follows that the dag size of the principal typing of any `let`-free ML expression is linear.

Proposition 3.3 *Given a* `let`*-free expression M of length n, there is a linear time algorithm which computes a dag representing the principal typing of M, if it exists, and returns* untypable *otherwise. If it exists, the principal typing of M has length $2^{O(n)}$ and dag size $O(n)$.*

Proof. For the `let`-free case the principal typing, if it exists, can be computed using only the Curry typing rules of the Appendix. An algorithm for this computation is the algorithm PT of the Appendix, without its second parameter A and without its last case. Assume that, in the absence of `let`, PT maintains type expressions using dags and that it uses a polynomial time subroutine for unification. It is easy to see that this is a polynomial time algorithm. It makes a number of calls to unification that is polynomial in the size of the input, and also, using the dag representation of most general unifiers, the parameters passed to these calls are polynomial-size bounded.

Even if unification of dags is linear time, this algorithm makes multiple calls to unification, and thus is not necessarily linear time. Let us argue that a nonrecursive version of this algorithm can be made to run in linear time, because it only involves one call to the unification subroutine.

To achieve linear time one may proceed as follows: (1) parse the `let`-free lambda term and rename all its bound variables to be distinct, (2) for each node of the parse tree pick a type variable denoting the principle type of the lambda term rooted at that node, (3) write constraints for each node as equations among type expressions – there are three kinds depending on whether the node is an application, an abstraction or a variable, (4) solve all these constraints simultaneously.

The Curry typing rules are "syntax directed" [GR88,Mit88]. Thus, by structural induction on lambda terms, one can show that the above nonrecursive algorithm is correct. Steps (1–3) are clearly linear time. By [PW78], step (4) can be done in linear time. ∎

3.2 Type size with "let"

Before discussing expressions with large types, it may be helpful to review the behavior of the ML typing algorithm on `let` expressions. To simplify matters, we will consider `let` $f = M$ `in` N with M closed. Essentially, this expression is typed by first computing the principal type of M. This type will generally contain type variables which are used as "place holders" for arbitrary types. In typing the body N, each occurrence of f is given a copy of the principal type of M *with all type variables renamed to be different from those used in every other type.* (Something similar but slightly more complicated is done when M is not closed.) Type inference then proceeds as usual, using different types for different occurrences of f. Because variables are renamed, the number of type variables involved in the principal type of an expression may be exponential. The following example is due to Mitchell Wand and, independently, to Peter Buneman.

Example 3.4 *Consider the expression*

$$\texttt{let } x = M \texttt{ in } \langle x, x \rangle$$

where M is closed with principal type σ. The principal type of this expression is $\sigma' \times \sigma''$, where σ' and σ'' are copies of σ with type variables renamed differently in each case. Thus, unlike $(\lambda x.\langle x, x\rangle)M$, not only is the type twice as long as σ, but the type has twice as many type variables. For this reason, even the dag representation for the type of the expression with `let` *is twice as large as the dag representation for the type of M. By nesting declarations of pairs, we can produce expressions*

$W_n ::=$
$\quad$ let $x_0 = M$ in
$\quad\quad$ let $x_1 = \langle x_0, x_0 \rangle$ in
$\quad\quad\quad \ldots$
$\quad\quad\quad$ let $x_n = \langle x_{n-1}, x_{n-1} \rangle$ in x_n

with n nested declarations whose principal types have $2^{\Omega(n)}$ type variables. Consequently, the dag representations of these types will have $2^{\Omega(n)}$ nodes.

It is worth mentioning that although the dag representation of the principal type for the expression in Example 3.4 has exponential size, other types for this expression have smaller dag representations. In particular, consider the instance obtained by applying a substitution which replaces all type variables with a single variable t. Since all subexpressions of the resulting type share the same type variable, this produces a typing with linear size dag representation. However, there are other expressions with principal types that are doubly-exponential when written out symbolically, and such that the dag representation of *any* type must have at least exponential size.

Example 3.5 *Recall that the expression $P ::= \lambda x.\langle x, x \rangle$ from Example 3.1 doubles the length of the principal type of its argument. Consequently, the n-fold composition of P (with itself) increases the length of the principal type by 2^n. Using nested* let*'s, we can define the 2^n-fold composition of P using an expression of length n. This gives us an expression whose principal type has double-exponential length, and exponential dag size. Since there is only a single type variable in the principal type, any substitution instance of the principal type also has exponential dag size. To see how this works, consider the expression V_n defined as follows.*

$V_n ::=$
$\quad$ let $x_1 = \lambda x.\langle x, x \rangle$ in
$\quad\quad$ let $x_2 = \lambda y.x_1(x_1 y)$ in
$\quad\quad\quad \ldots$
$\quad\quad\quad$ let $x_n = \lambda y.x_{n-1}(x_{n-1} y)$ in $x_n(\lambda z.z)$

To write the principal type of this expression simply, let us use the notation $\tau^{[n]}$ for the n-ary product defined inductively by $\tau^{[1]} ::= \tau$ and $\tau^{[n+1]} ::= (\tau^{[n]}) \times (\tau^{[n]})$. It is easy to see that $\tau^{[n]}$ has $2^{\Omega(n)}$ symbols. By examining the expression V_n and tracing the behavior of the ML typing algorithm, we can see that $x_1 \equiv P$ has principal type $t \to t^{[2]}$, and for each $i > 0$ the principal type of x_i has type $t \to t^{[2^n]}$. Consequently, the principal type of the entire expression V_n is $(t \to t)^{[2^n]}$, which has $2^{2^{\Omega(n)}}$ symbols. Since a dag representation can reduce this by at most one exponential, the dag size of the principal type is $2^{\Omega(n)}$.

Proposition 3.6 *For arbitrarily large n, there exist closed expressions of length n whose principal types have length $2^{2^{\Omega(n)}}$, dag size $2^{\Omega(n)}$, and $2^{\Omega(n)}$ distinct type variables. Furthermore, every instance of the principal type must have dag size $2^{\Omega(n)}$.*

Proof. For any $k \geq 1$, we may construct an expression W_k as in Example 3.4 whose principal types has exponentially many type variables, and an expression V_k as in Example 3.5 satisfying the remaining conditions. Therefore the expression $\langle W_k, V_k \rangle$ suffices to prove the proposition. ∎

3.3 Upper bound with "let"

One way to type a core ML expression is simply to reduce to `let` normal form and then use a polynomial time algorithm, e.g., PT of the Appendix or the linear time algorithm of Proposition 3.3. The main reason this method works properly is that typing is generally preserved by `let` reduction, as described in the following lemma. The one exception occurs when we reduce an expression $\texttt{let}\ x = M\ \texttt{in}\ N$ where x does not occur free in N. However, this case may be eliminated by a simple syntactic transformation.

Lemma 3.7 *Consider an expression* $\texttt{let}\ x = M\ \texttt{in}\ N$ *and the result* $[M/x]N$ *of* `let` *reduction. If M is typable or x occurs free in N, then these expressions have precisely the same typings, and hence the same principal typing.*

The proof of this is immediate, by inspection of the inference rule for `let` given in the Appendix.

Lemma 3.8 *A core ML expression of the form* $\texttt{let}\ x = M\ \texttt{in}\ N$ *has precisely the same typings as the expression* $\texttt{let}\ x = M\ \texttt{in}\ (\lambda y.N)x$, *provided y not free in N. In particular, the two expressions have the same principal typing.*

Note that in the second expression $\texttt{let}\ x = M\ \texttt{in}\ (\lambda y.N)x$ of Lemma 3.8, the bound variable x occurs free in $(\lambda y.N)x$, and so by Lemma 3.7 the principal type of this term is preserved by `let` reduction.

Proof. We examine the behavior of algorithm PT, given in the Appendix, on the two expressions

$$\begin{aligned} L_1 &::= \texttt{let}\ x = M\ \texttt{in}\ N, \\ L_2 &::= \texttt{let}\ x = M\ \texttt{in}\ (\lambda y.N)x. \end{aligned}$$

We must show that either $PT(L_1, A)$ and $PT(L_2, A)$ both fail, or produce the same principal typing. Suppose that $PT(L_1, A)$ succeeds. Following the notation of the Appendix, let

$$\begin{array}{lcl} \Gamma_1 \rhd M : \sigma & = & PT(M, A) \\ A' & = & A \cup \{x \mapsto \Gamma_1 \rhd M : \sigma\} \\ \Gamma_2 \rhd N : \tau & = & PT(N, A') \\ S & = & UNIFY(\{\, \alpha = \beta \mid y{:}\alpha \in \Gamma_1 \text{ and } y{:}\beta \in \Gamma_2 \,\}) \end{array}$$

and recall that the principal typing produced is

$$S\Gamma_1 \cup S\Gamma_2 \rhd \texttt{let}\ x = M\ \texttt{in}\ N : S\tau$$

We will show that $PT(L_2, A)$ succeeds with the same result.

By examining the recursive calls in $PT(L_2, A)$, we see that

$$PT(\lambda y.N, A') = \Gamma_2 \rhd \lambda y.N : s \rightarrow \tau$$

with s fresh, since y does not occur free in N. By definition of A' and the variable case of PT, we also have

$$PT(x, A') = \Gamma_1 \rhd x : \sigma$$

and so by inspection of the application case of PT,

$$PT((\lambda y.N)x, A') = S'\Gamma_1 \cup S'\Gamma_2 \rhd (\lambda y.N)x : S't$$

where $S' = UNIFY(\{\alpha = \beta \mid y{:}\alpha \in \Gamma_1 \text{ and } y{:}\beta \in \Gamma_2\} \cup \{s{\to}\tau = \sigma{\to}t\})$ and t is fresh. Since S exists (by hypothesis), and both s and t are fresh type variables chosen by PT, the substitution S' is simply S, with the additional mapping $s \mapsto S\sigma$ and $t \mapsto S\tau$. Thus $PT(L_2, A)$ succeeds with the desired typing.

The remaining case to consider is when $PT(L_1, A)$ fails. If so, than one of the recursive calls to PT must fail, or the call to $UNIFY$ fails. But since the same calculations are involved in $PT(L_2, A)$, this must fail also. This completes the proof. ∎

In general, the length of a core ML expression may increase exponentially as a result of `let`-reduction. We can give a more precise description of the increase by considering the way that `let`'s occur. To be precise, we define the `let`-*depth,* $\ell d(M)$, *of* M inductively as follows.

$$\begin{array}{lcl} \ell d(x) & = & 1 \\ \ell d(MN) & = & max\{\ell d(M), \ell d(N)\} \\ \ell d(\lambda x.M) & = & \ell d(M) \\ \ell d(\texttt{let } x = M \texttt{ in } N) & = & \ell d(M) + \ell d(N)\} \end{array}$$

The depth of `let` $x = M$ `in` N is additive since when we reduce to $[M/x]N$, expressions nested to depth $\ell d(M)$ in M may become nested to depth $\ell d(M) + \ell d(N)$ by substitution into N. In the common special case that M is `let`-free, the `let`-depth of `let` $x = M$ `in` N is $1 + \ell d(N)$.

Lemma 3.9 *Let N be the* `let` *normal form of an ML expression M. Then $|N| = 2^{O(\ell \log_2 |M|)}$, where ℓ is the* `let`*-depth of M.*

Proof. The proof is a straightforward induction on the syntax of expressions. The lemma is trivially true for a variable x. For an application MN, the `let` normal form is $M'N'$, where M' and N' are the `let` normal forms of M and N, respectively. By the inductive hypothesis, we have

$$|M'N'| = 2^{O(\ell d(M) \log_2 |M|)} + 2^{O(\ell d(N) \log_2 |N|)} = 2^{O(max\{\ell d(M), \ell d(N)\} \log_2 |MN|)}$$

The lambda abstraction case is similarly straightforward. The remaining case is `let` $x = M$ `in` N. Remember that the `let` normal form is unique, by Proposition 2.1, so we may reduce the expression in any order. The simplest way to estimate length is to first reduce M to `let` normal form M' and similarly reduce N to N'. The `let` normal form of the entire expression is then obtained by substitution $[M'/x]N'$. Using the inductive hypothesis, we have

$$\begin{array}{rcl} |[M'/x]N'| & = & 2^{O(\ell d(M) \log_2 |M|)} 2^{O(\ell d(N) \log_2 |N|)} \\ & = & 2^{O((\ell d(M) + \ell d(N)) max\{\log_2 |M|, \log_2 |N|\})} \\ & = & 2^{O(\ell d(\texttt{let } x=M \texttt{ in } N) \log_2 |\texttt{let } x=M \texttt{ in } N|)} \end{array}$$

∎

Theorem 3.10 *There is an algorithm which decides whether any given core ML expression M has a provable typing in time $2^{O(\ell \log_2 |M|)}$, where ℓ is the* `let`*-depth of M. When M is typable, this algorithm yields a dag representing the principal typing.*

Corollary 3.11 *If M is a core ML expression of length n, then a principal typing of M has length at most doubly-exponential in n, and dag size $2^{O(n)}$.*

Corollary 3.12 *If we restrict our attention to ML expressions with* `let`*-depth ℓ, for some fixed ℓ, then we can determine typability, and compute principal typings, in polynomial time.*

4 Extended type expressions and polymorphic unification

4.1 Extended type expressions

The extended type expressions include a polymorphic `let` declaration which resembles the polymorphic `let` in ML expressions. Formally, the *extended type expressions* are defined as follows,

$$\sigma \quad ::= \quad t \mid \sigma \to \sigma \mid \texttt{let}\ t = \sigma\ \texttt{in}\ \sigma$$

where t may be any type variable. There is an important difference between the `let` in extended type expressions and in core ML expressions. The simplest way to think of extended type expression $\texttt{let}\ t = \sigma\ \texttt{in}\ \tau$ is as: *an abbreviation for the expansion* $[[\sigma/t]]\tau$ *obtained by replacing each occurrence of* t *in* τ *with a different alphabetic variant of* σ.

More specifically, if τ has k occurrences of t, then we choose k alphabetic variants $\sigma_1, \ldots, \sigma_k$ of σ such that no type variable appears in more than one of the expressions $\sigma_1, \ldots, \sigma_k, \tau$. Then, we replace the k-th occurrence of t in τ by σ_k. For example, $\texttt{let}\ t = r \to r\ \texttt{in}\ t \to t$ may be understood as an abbreviation for the expansion

$$[[r \to r/t]](t \to t) \quad = \quad (r' \to r') \to (r'' \to r'').$$

A precise inductive definition requires some "book-keeping" mechanism to guarantee that all alphabetic variants use distinct type variables, but is otherwise straightforward.

Note that as described, $[[\sigma/t]]\tau$ is only defined up to renamings of type variables. However, renaming type variables does not effect the unifiability of expressions (provided distinct variables are not identified). By Proposition 2.1, adapted to type expressions, every extended type expression may be expanded to a let-free ordinary type expression, unique modulo renamings of type variables introduced by expansion. We write $expand(\sigma)$ for any type expression obtained from σ by expanding all `let`'s.

Since we consider a type expression equivalent to its expansion, extended type expressions may be viewed as a technical device for obtaining succinct representations of ordinary type expressions. However, the obvious counting argument shows that not all ordinary type expressions may be represented succinctly by extended expressions.

Example 4.1 *Recall the following expression from Example 3.4.*

$$\begin{array}{l}\texttt{let}\ x_0 = M\ \texttt{in} \\ \quad \texttt{let}\ x_1 = \langle x_0, x_0 \rangle\ \texttt{in} \\ \qquad \ldots \\ \qquad \texttt{let}\ x_k = \langle x_{k-1}, x_{k-1} \rangle\ \texttt{in}\ x_k\end{array}$$

Using extended type expressions, its principal type may be written as

$$\begin{array}{l}\texttt{let}\ t_0 = \sigma\ \texttt{in} \\ \quad \texttt{let}\ t_1 = t_0 \times t_0\ \texttt{in} \\ \qquad \ldots \\ \qquad \texttt{let}\ t_k = t_{k-1} \times t_{k-1}\ \texttt{in}\ t_k\end{array}$$

where we assume M *has principal type* σ. *The extended type expression is linear in the size of the expression (assuming* σ *is linear in the size of* M*), but when we expand all* `let`*'s, we obtain the exponential-size type described in Example 3.4. Because of the way type variables are renamed in expansion, the expanded type will have exponentially many type variables.*

4.2 Polymorphic unification

Polymorphic unification is the unification problem for extended type expressions. More precisely, a substitution S *unifies extended type expressions* σ *and* τ if $S\,expand(\sigma) = S\,expand(\tau)$.

Theorem 4.2 *Given two extended type expressions* σ *and* τ*, it is PSPACE hard to determine whether* σ *and* τ *are unifiable.*

Proof. The proof is by reduction from quantified propositional formulas. We assume we are given a formula

$$\phi \equiv Q_1 P_1 \ldots Q_n P_n . \psi$$

where $Q_1, \ldots, Q_n$ are quantifiers $\forall$ or $\exists$, symbols $P_1, \ldots, P_n$ are propositional variables, and ψ is a matrix

$$\psi \equiv C_1 \wedge \ldots \wedge C_m$$

of disjuncts of positive and negated variables. In other words, each C_i has the form

$$C_i \equiv P_{i,1} \vee \ldots \vee P_{i,j} \vee \neg P_{i,j+1} \vee \ldots \vee \neg P_{i,l}$$

for some sequence of propositional variables $P_{i,1}, \ldots, P_{i,l}$. It is well-known that determining the validity of such formulas is PSPACE complete [GJ79,SM73].

Given ϕ (as described above) we will construct extended type expressions σ and τ which are unifiable iff ϕ is not valid.

NP-hardness: We first give the construction when all quantifiers are $\exists$.

Expression σ: is an extended expression which, when expanded and viewed as a directed acyclic graph, looks like a full binary "tree" of height n, with distinct subgraphs $\alpha_1, \ldots, \alpha_{2^n}$ placed at the "leaves." (Strictly speaking, the graph is not a tree since the α's will not generally be trees, this is why we use the notation "tree" and "leaves".) Subgraph α_i encodes the truth values of conjuncts $C_1, \ldots, C_m$ at the i-th truth assignment to variables $P_1, \ldots, P_n$. More specifically, each α_i is constructed to have $2m$ leaves, two for each conjunct. If conjunct C_j is true at the i-th assignment, then leaves $2j-1$ and $2j$ of α_i will be the same node; otherwise, all leaves are distinct.

Expression τ: depends only on the number of propositional variables and the number of conjuncts. When expanded and viewed as a dag, τ also looks like a full binary "tree" of height n, with copies of a certain test graph β placed at each "leaf." Essentially, when we attempt to unify σ and τ, the i-th copy of β tests to see whether α_i encodes a satisfying assignment for ψ.

Since it seems difficult to construct the α_i's directly, we construct the entire type expression σ by unifying n expressions $\sigma_1, \ldots, \sigma_n$, with σ_i encoding the dependence of ψ on the value of the i-the propositional variable P_i. A minor technical detail is that we must actually produce a unification problem that causes σ to be built from $\sigma_1, \ldots, \sigma_n$ in the process of testing for unifiability with τ. However, this is easily arranged by standard techniques [PW78,DKM84]. By arranging the σ_i's as appropriate children of two special root nodes, we can obtain an expression $\sigma = Unify(\sigma_1, \ldots, \sigma_k)$ that encodes an enumeration of all truth assignments to the propositional variables, and the resulting truth values of $C_1, \ldots, C_m$.

Expression σ_i: is built as follows. We begin with two expressions T_i and F_i. Intuitively, T_i tells what happens to the matrix ψ when $P_i = True$, and F_i encodes the result of setting $P_i = False$. In graphical terms, we think of T_i as a root with $2m$ children, one pair of children for each conjunct of the propositional formula. If setting $P_i = True$ guarantees that $C_j = True$, then children $2j-1$

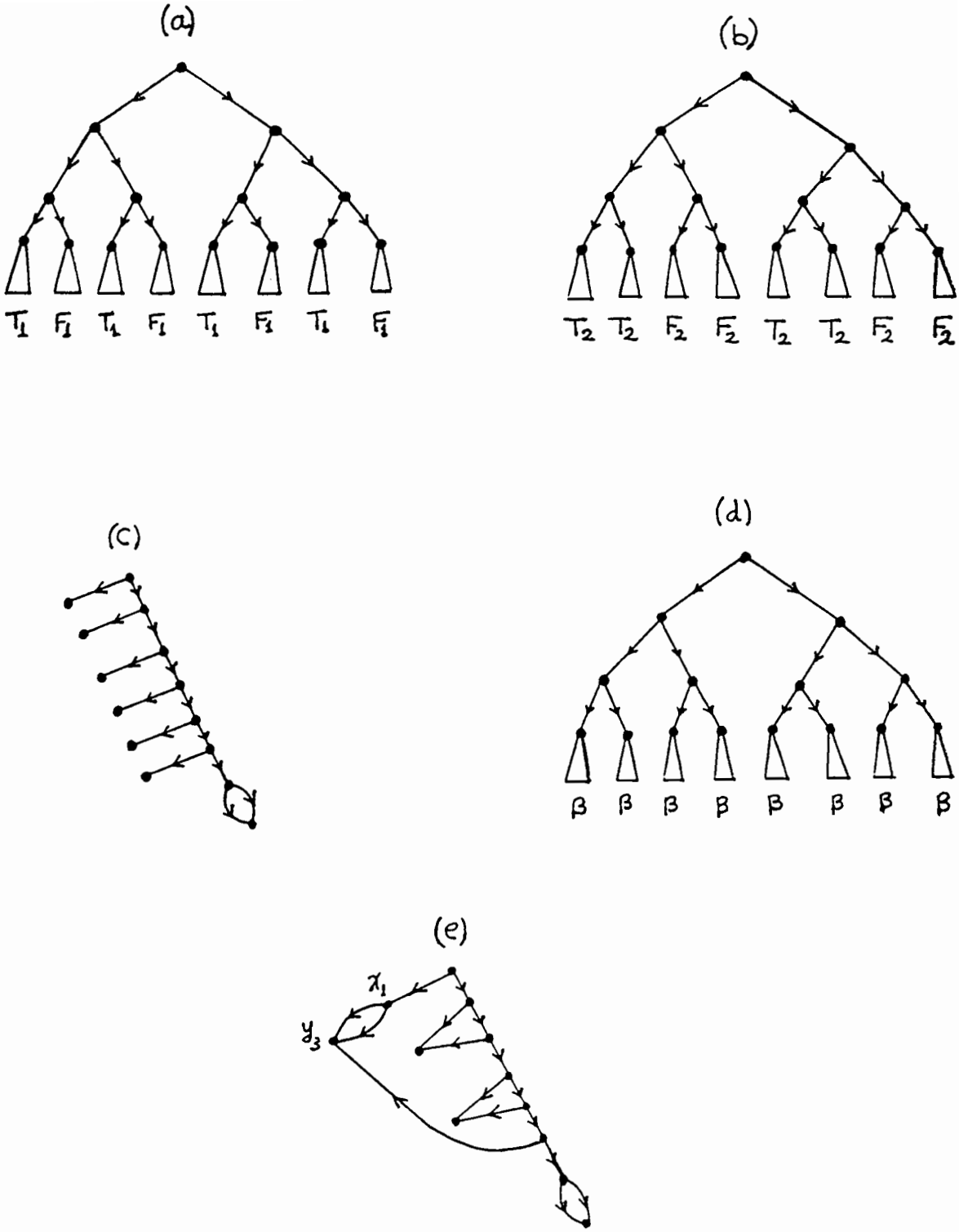

Figure 1: (a) σ_1 expansion, (b) σ_2 expansion, (c) T_i, F_i skeleton, (d) τ expansion, (e) β.

and $2j$ of T_i are the same leaf. Otherwise, we let children $2j-1$ and $2j$ be distinct. To be more precise, since we do not have $2m$-ary function symbols, we let T_i have the form

$$T_i \equiv s_{i,1} \rightarrow t_{i,1} \rightarrow s_{i,2} \rightarrow t_{i,2} \ldots \rightarrow s_{i,m} \rightarrow t_{i,m} \rightarrow r \rightarrow r$$

where $s_{i,j} \equiv t_{i,j}$ iff setting $P_i = \mathit{True}$ forces clause $C_i = \mathit{True}$. The extra variable r is included to simplify a technical problem in the proof of Theorem 5.1, which follows the structure of this proof. Expression F_i is defined similarly. Once we have T_i and F_i, we construct σ_i by a nested sequence of `let`'s. For $i = 1$, the graph σ_1 is constructed as follows.

$\sigma_1 ::=$
  `let` $t_1 = T_1 \rightarrow F_1$ `in`
    `let` $t_2 = t_1 \rightarrow t_1$ `in`
      ...
      `let` $t_n = t_{n-1} \rightarrow t_{n-1}$ `in` t_n

When expanded, σ_1 becomes an expression whose dag representation is a binary tree with "leaves" alternating between dags for T_1 and F_1. For $i > 1$, we use a similar construction, but arrange the nesting of `let`'s so that the pattern of T_i's and F_i's differs from all other patterns.

In Figure 1.a we illustrate what σ_1 would look like after expansion and in Figure 1.b what σ_2 would look like. The *skeleton* of the expressions for T_i, F_i is illustarted in Figure 1.c, for $m = 3$. Depending on the effect of variable P_i on the truth value of clauses certain pairs of leaves in the *comb* like part of these expressions are identified.

The expression τ is constructed by a similar nesting of `let`'s, placing an m-ary AND gate, as in [DKM84], at each leaf, so that if all the conjuncts are true at any assignment, unification fails. In Figure 1.d we illustrate τ and in Figure 1.e give the expression for the AND gate β.

It is clear from the structure of the AND gate that unifying it with a collection of n T_i's and F_j's will fail iff *all* pairs of nodes along the comb like part of these expressions are identified. The failure is forced by failing an acyclicity test, [PW78], at the output of the AND gate, i.e., at the pair (x_1, y_m). Thus, we have unifiability iff there is a no satisfying assignment for the formula ψ iff ϕ is not valid. This concludes the proof for the special case, in which all quantifiers are existential.

PSPACE-hardness: In order to treat the $\forall$ and $\exists$ quantifiers we must construct τ more carefully. In addition to testing which assignments satisfy the matrix ψ, we must AND and OR the results of these tests properly. Here we outline the modifications to the NP-hardness proof. The full details are listed in the proof of Theorem 5.1, which also contains realizations by core ML expressions. The AND and OR expressions used in this construction are both from [DKM84].

The PSPACE-hardness proof is similar in structure with the NP-hardness proof, except that each "subtree" of the σ_i's and τ has two special nodes encoding the output of that "subtree". For the σ_i's the outputs are not connected to anything. They exist only so that these expressions may unify with τ, when ϕ is not valid.

The critical modification is adding structure to the internal nodes of the full binary "tree" τ. Nodes at depth i are transformed into binary AND or into OR gates, depending on the quantifier used in ϕ at depth i. The gates can be connected so that, within τ the output of each subtree is a Boolean combination of the tests at the leaves of this "subtree".

In Figure 2.a we illustrate the modified skeletons of T_i, F_i with output (u, v). They are just as the previous skeletons with one extra pair of nodes in the comb. In Figure 2.b we illustrate the modified AND gate. Instead of a part of the gate that forces an acyclicity failure we have output

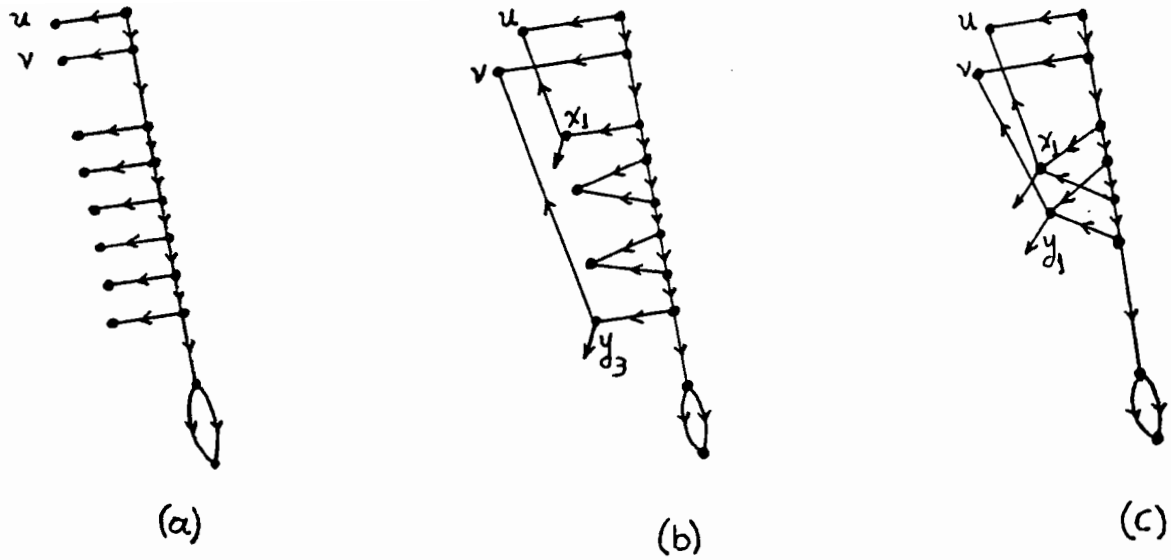

Figure 2: (a) skeletons of T_i, F_i with output u, v, (b) m-ary AND gate with output u, v, (c) binary OR gate with output u, v.

nodes (u, v). This is an m-ary gate. The binary AND gate is a special case of Figure 2.b. The binary OR gate is illustrated in Figure 2.c.

A simple inductive argument suffices to show that: in unifying σ and τ the output nodes of the "tree" τ unify iff ϕ is not valid. A failure can now be forced by failing the acyclicity test at these nodes. Thus, σ and τ unify iff ϕ is not valid. ∎

5 Lower bound for ML typing

Theorem 5.1 *It is PSPACE hard to determine whether a given core ML expression is typable.*

Proof. The main idea of the proof is to re-trace the proof of Theorem 4.2, showing that all extended type expressions may be defined as principal types of ML expressions. By this method, we show that for any type expressions σ and τ constructed as in the proof of Theorem 4.2, there is a core ML expression which is typable iff σ and τ are unifiable. The expressions we use at intermediate steps in the proof will have approximately the same length as the extended expressions for their principal types.

However, there is one technical complication that should be mentioned at the outset. It is well-known that only types which resemble propositional tautologies can be the types of lambda terms [How80]. In fact, given a type expression, deciding if there exists some lambda term such that the expression is a type for this term is a nontrivial task (it is shown PSPACE-complete in [Sta79]). So, the realization of extended type expressions as types of lambda terms can be involved.

NP-hardness: We first realize T_i, F_i (Figure 1.c) and the AND gate β (Figure 1.e). We then show how to construct τ (Figure 1.d) and the σ_i's (Figure 1.ab) and how to force their unification.

We describe a lambda term M_i with principal type T_i. Recall that T_i has the form

$$T_i \equiv s_{i,1} \rightarrow t_{i,1} \rightarrow s_{i,2} \rightarrow t_{i,2} \ldots \rightarrow s_{i,m} \rightarrow t_{i,m} \rightarrow r \rightarrow r,$$

where $\rightarrow$ associates to the right. Variables $s_{i,j}$ and $t_{i,j}$ will be identical in T_i iff setting $P_i = \mathit{True}$ forces clause $C_j = \mathit{True}$. For each T_i, we build a lambda term M_i of the form

$$\begin{aligned} M_i \equiv\ & \lambda x_1.\lambda y_1.\ \ldots\ \lambda x_m.\lambda y_m. \\ & \lambda z.\, K\, z\, \langle Eq(x_{i_1}, y_{i_1}), \ldots, Eq(x_{i_k}, y_{i_k})\rangle \end{aligned}$$

where $K = \lambda u.\lambda v.u$ and the subexpression $Eq(x,y)$ forces variables x and y to have the same type. Thus, we make x_j, y_j have the same types iff setting P_i=true forces C_j=true. To be concrete, we can take

$$Eq(x,y) \equiv \lambda z.\, K(zx)(zy),$$

since typing this expression forces the types of x and y to be unified. To see this, reason as follows. Since $K: s\rightarrow t\rightarrow s$, the only constraint on x and y is that we unify their types and let z: *Unify*((type of x), (type of y))$\rightarrow t$ for some fresh t (we use : as an abbreviation for is-principal-type-of). The types of the $Eq(x_{i_j}, y_{i_j})$ are irrelevant, since K discards its second argument.

The expression N_i with principal type F_i is defined similarly.

The expression A with principal type β is defined as follows,

$$\begin{aligned} A \equiv\ & \lambda x_1.\lambda y_1.\ \ldots\ \lambda x_m.\lambda y_m. \\ & \lambda z.\, K\, z\, \langle Eq(y_1, x_2), Eq(y_2, x_3), \ldots, Eq(y_{m-1}, x_m), B\rangle \end{aligned}$$

where it B is the term $\lambda w.K(w(\lambda z.z))\langle wx_1, x_1 y_m\rangle$.

This term forces the following. (1) w:$(t\rightarrow t)\rightarrow s$ where s,t are fresh type variables, (2) x_1:$t\rightarrow t$, (3) y_m:t. This makes it impossible to unify the types of x_1 and y_m, the outputs of the AND gate. So unifying the output nodes of β forces a failure of the acyclicity test.

The final part of the construction is the way we build depth-n "trees" with T_i, F_i, β as "leaves". Assume we have already declared in a core ML expression,

`let` $l = M$ `in let` $r = N$ `in` ... `end end`

where l and r are expressions whose principal types are left L and right R "subtrees" of some internal node. We construct a term whose principal type is $L\rightarrow R\rightarrow t\rightarrow t$ as follows (see Figure 3.a).

`let` *root* $= \lambda x.\lambda y.\lambda z.\, K\, z\, \langle Eq(x,l), Eq(y,r)\rangle$

The entire expression, which is typable iff ψ is satisfiable, is an ultimate nested sequence of `let`'s defining expressions $\Sigma_1, \ldots, \Sigma_n$ with principal types $\sigma_1, \ldots, \sigma_n$ and expression T with principal type τ. Inside this ultimate sequence of `let`'s, as the innermost expression, we put,

$$\lambda z.\, K\, z\, \langle Eq(z, \Sigma_1), Eq(z, \Sigma_2), \ldots, Eq(z, \Sigma_n), Eq(z, T)\rangle$$

This forces the unification of $\sigma_1, \ldots, \sigma_n$ and of τ and completes the NP-hardness proof.

PSPACE-hardness: It suffices to realize the appropriate T_i, F_i (Figure 2.a) the AND gate (Figure 2.b) the OR gate (Figure 2.c) and to explain the construction of the internal nodes of the various depth-n "trees".

The realization of T_i, F_i is as before, with two more lambda abstractions. The m-ary AND gate is realized as the principal type of the lambda term,

$$\begin{aligned} A \equiv\ & \lambda u.\lambda v.\lambda x_1.\lambda y_1.\ \ldots\ \lambda x_m.\lambda y_m. \\ & \lambda z.\, K\, z\, \langle Eq(y_1, x_2), Eq(y_2, x_3), \ldots, Eq(y_{m-1}, x_m), x_1 u, y_m v\rangle \end{aligned}$$

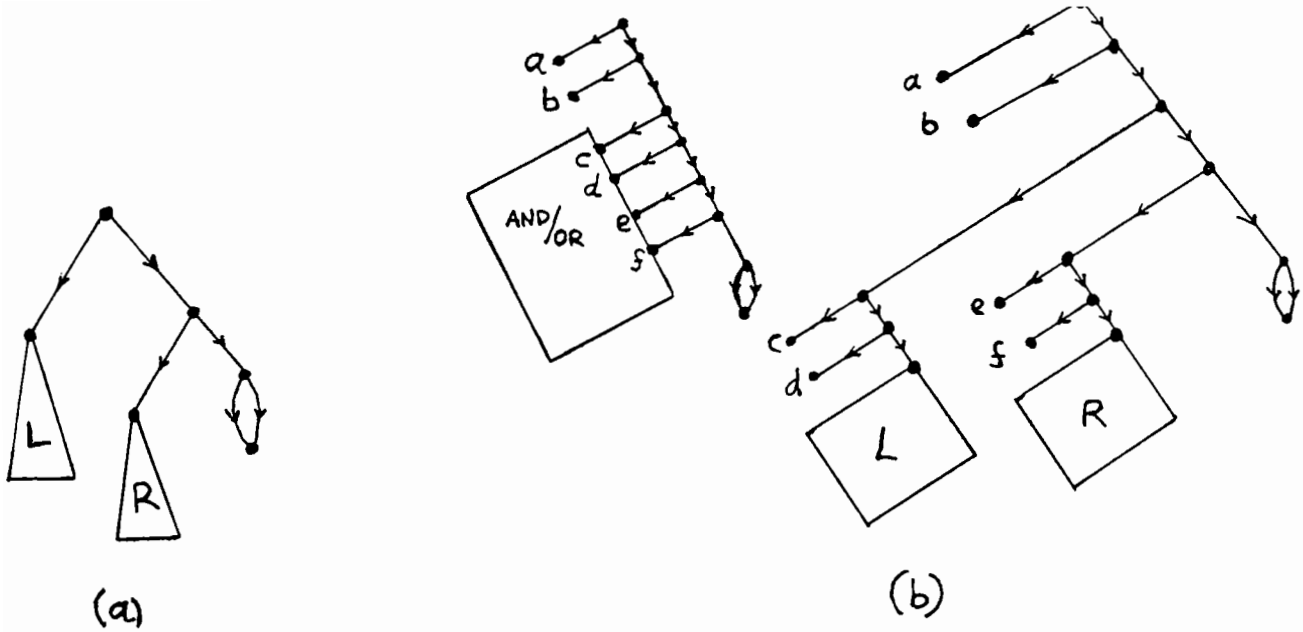

Figure 3: (a) internal node for NP-hardness, (b) internal node for PSPACE-hardness.

and the binary AND gate is a special case. The binary OR gate is realized as the principal type of the lambda term,

$$\begin{aligned} O \equiv\ & \lambda u.\lambda v.\lambda x_1.\lambda y_1.\lambda x_2.\lambda y_2. \\ & \lambda z.\, K\, z\, \langle Eq(x_1, x_2), Eq(y_1, y_2), x_1 u, y_1 v\rangle \end{aligned}$$

We now present the construction of internal nodes for the various depth-n "trees". The internal nodes construction for the NP-hardness proof, illustrated in Figure 3.a, is replaced by the construction illustrated in Figure 3.b. It is more complex for τ than for the σ_i's.

First we need an observation. If we have a term of the form,

$$\lambda g.\ldots.\lambda u.\lambda v.\lambda x_1.\lambda y_1.\lambda x_2.\lambda y_2.\ldots. Eq(g, AND)\ldots(g\, u\, v\, x_1\, y_1\, x_2\, y_2)$$

then g will have a type that is a "copy" of an AND gate (assuming AND is a let-bound name), and the types of u, v will be unified with the outputs and the types of x_1, y_1, x_2, y_2 will be unified with the inputs.

Realizing τ: each such internal node is constructed by combining a left subtree, right subtree and logical gate as in Figure 3.b. In order to prove correctness, we assume inductively that c, d give us the output of the left subtree and $e.f$ the output of the right subtree. These are connected to the inputs of a fresh AND or OR gate and the outputs of the logic gate are unified with the top two leaves of the resulting "subtree". Assuming we have let-bound names l and r for the left and right "subtrees", the resulting "subtree" will be the principal type of the expression IN below. By nesting a sequence of n `let`'s, one for each level of τ, and using this construction at each level, we can realize τ as the principal type of a core ML expression.

$$\begin{aligned} IN \equiv\ & \lambda u.\lambda v.\lambda x.\lambda y.\lambda z.\, K\, z \\ & (\lambda gate.\lambda left.\lambda right.\lambda a.\lambda b.\lambda c.\lambda d.\lambda e.\lambda f. \\ & \langle Eq(gate, AND/OR), Eq(left, l), Eq(right, r), \\ & (gate\, a\, b\, c\, d\, e\, f), Eq(u, a), Eq(v, b), \\ & (left\, c\, d), (right\, e\, f)\rangle) \end{aligned}$$

Realizing σ_i: the construction is similar with that of τ. In fact, it is simpler. All we have to do is replace the logic gates by "dummy" type variables. This is done by replacing the AND and OR in IN with $\lambda a.\lambda b.\lambda c.\lambda d.\lambda e.\lambda f.\lambda z.z$.

Forcing the unification of τ and the σ_i's is as in the NP-hardness part. This completes the proof. ∎

6 Conclusion and future directions

Milner's typing algorithm for ML is widely used and generally regarded as "efficient." However, in the worst case, it requires doubly-exponential time to produce its string output. We have seen that if we no longer require printing the type as a string, the algorithm could be modified to run in exponential time. However, recognizing the typable ML expressions is PSPACE hard. Therefore, it is unlikely the algorithm could be optimized further, unless P=PSPACE.

A basic intuition is that: `let` in type expressions acts as a succinctness primitive and leads to an increase in complexity (this is analogous to many problems from [SM73]). A remaining technical problem is to fully characterize this succinctness primitive, by closing the gap between our PSPACE lower bound and the deterministic exponential time upper bound.

One promising direction for further work is to apply the polymorphic unification technique described here to the related typing problems described in [GR88,Hen88,KTU88,Lei83,Mit88]. Since polymorphic unification is independent of ML syntax, we hope that this formulation will be useful in deriving additional lower bounds.

Since ML typing appears efficient in practice, it seems that worst-case typing problems must occur with very low frequency. There are two likely explanations. The first is that as noted in Corollary 3.12, ML typing depends primarily on the `let`-depth of terms, as opposed to their length. While programs such as the CAML compiler [CCM85] begin with a very long sequence of `let` declarations, it is possible that the actual chains of declarations which depend on each other nontrivially are relatively short. A second possibility is that although functions declared by `let` are typically polymorphic, it seems common to apply them to non-polymorphic arguments. This keeps the number of type variables from growing exponentially, and would therefore seem likely to reduce the complexity of typing. However, both of these explanations require further investigation.

Acknowledgements: We are grateful to Mitchell Wand and Peter Buneman for pointing out a number of interesting examples, and thank Gérard Huet and David MacQueen for discussions regarding the relevance of our lower bound to ML practice. The first author would also like to thank Ken Perry for many helpful discussions, and acknowledge the support of the IBM T.J. Watson Research Center.

References

[AHU83] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[Bar84] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North Holland, 1984.

[CCM85] G. Cousineau, P.L. Curien, and M. Mauny. The categorical abstract machine. In *IFIP Int'l Conf. on Functional Programming and Computer Architecture, Nancy, Lecture Notes in Computer Science 201*, Springer-Verlag, 1985.

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

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

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

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., 1979.

[GMW79] M.J. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[GR88] P. Giannini and S. Ronchi Della Rocca. Characterization of typings in polymorphic type discipline. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 61–71, July 1988.

[Hen88] F. Henglein. Type inference and semi-unification. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 184–197, July 1988.

[How80] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490, Academic Press, 1980.

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

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

[MH88] J.C. Mitchell and R. Harper. The essence of ML. In *Proc. 15-th ACM Symp. on Principles of Programming Languages*, pages 28–46, January 1988.

[Mil78] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17, 1978. pages 348-375.

[Mil85] R. Milner. The Standard ML core language. *Polymorphism*, 2(2), 1985. 28 pages. An earlier version appeared in Proc. 1984 ACM Symp. on Lisp and Functional Programming.

[Mit88] J.C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2/3), 1988.

[Mit89] J.C. Mitchell. Type systems for programming languages. To appear as a chapter in *Handbook of Theoretical Computer Science*, van Leeuwen et. al. eds. North-Holland, 1989.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. on Prog. Lang. and Systems*, 4(2), Feb. 1982.

[PW78] M.S. Paterson and M.N. Wegman. Linear unification. *JCSS*, 16, 1978. pages 158–167.

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

[Sta79] R. Statman. Intuitionistic propositional logic is polynomial-space complete. *TCS*, 9, 1979, pages 67–72.

[SM73] L.J. Stockmeyer and A.R. Meyer. Word problems requiring exponential time. In *Proc. 5-th ACM Symp. on Theory of Computing*, pages 1–9, 1973.

[Tur85] D.A. Turner. Miranda: a non-strict functional language with polymorphic types. In *IFIP Int'l Conf. on Functional Programming and Computer Architecture, Nancy, Lecture Notes in Computer Science 201*, Springer-Verlag, 1985.

Appendix. Type inference rules and algorithm

We summarize an inference system and typing algorithm for core ML. Both are based on [DM82, Mil78], but presented in a manner which simplifies our analysis. For more details on this presentation we refer to [Mit89]. Typing assertions about core ML expressions without `let` may be proved using the following axiom and inference rules.

$$(var) \qquad x{:}\sigma \triangleright x : \sigma$$

$$(abs) \qquad \frac{\Gamma, x{:}\sigma \triangleright M : \tau}{\Gamma \triangleright (\lambda x.M) : \sigma \rightarrow \tau}$$

$$(app) \qquad \frac{\Gamma \triangleright M : \sigma \rightarrow \tau, \quad \Gamma \triangleright N : \sigma}{\Gamma \triangleright MN : \tau}$$

$$(add\ hyp) \qquad \frac{\Gamma \triangleright M : \sigma}{\Gamma, x{:}\tau \triangleright M : \sigma}\ , x \text{ not in } \Gamma$$

These rules are often called the *Curry typing rules*, after H.B Curry [CF58].

Typing with `let` may be characterized by adding the following inference rule, which uses substitution to capture the polymorphism of `let`.

$$(\texttt{let}) \qquad \frac{\Gamma \triangleright M : \sigma, \ \Gamma \triangleright [M/x]N : \tau}{\Gamma \triangleright \texttt{let } x = M \texttt{ in } N : \tau}$$

If we ignore the hypothesis about M, then this rule allows us to type `let` $x = M$ `in` N by typing the result $[M/x]N$ of `let` reduction. Polymorphism arises from the possibility of inferring different types for the different occurrences of M. The assumption about M in this rule is included to cover the special case when x does not appear in N. In general, the ML type checker only accepts `let` $x = M$ `in` N if M is well-typed. When x occurs in N, it suffices to find a type for $[M/x]N$. But when x does not occur in N, we need the additional condition in the antecedent of the rule.

The algorithm PT given below in Figure 4 computes a principal typing for any typable ML term. The algorithm has two arguments, a term to be typed, and an environment mapping variables to typing assertions. The purpose of the environment is to handle `let`-bound variables. The algorithm is written using an applicative, pattern-matching notation resembling the programming language Standard ML. It is assumed that: "the input to PT is a program with all bound variables renamed to be distinct". This is an operation that is commonly done in lexical analysis prior to parsing.

Algorithm PT may *fail* in the application or `let` case if the call to *UNIFY* fails. We can prove that if $PT(M, \emptyset)$ succeeds, then it produces a provable typing for M.

Theorem 6.1 *If $PT(M, \emptyset) = \Gamma \triangleright M : \tau$, then $\Gamma \triangleright M : \tau$ is a provable typing statement.*

It follows, by Proposition 2.2, that every instance of $PT(M, \emptyset)$ is provable. Conversely, every provable typing for M is an instance of $PT(M, \emptyset)$.

Theorem 6.2 *Suppose $\Gamma \triangleright M : \tau$ is a provable typing. Then $PT(M, \emptyset)$ succeeds and produces a typing with $\Gamma \triangleright M : \tau$ as an instance.*

$PT(x, A)$ = **if** $A(x) = \Gamma \triangleright M : \sigma$ **then** $\Gamma \triangleright x : \sigma$
else $\{x{:}t\} \triangleright x : t$

$PT(MN, A)$ =
let
$\Gamma_1 \triangleright M : \sigma = PT(M, A)$
$\Gamma_2 \triangleright N : \tau = PT(N, A)$,
with type variables renamed to be disjoint from those in $PT(M, A)$
$S = UNIFY(\{\, \alpha = \beta \mid x{:}\alpha \in \Gamma_1 \text{ and } x{:}\beta \in \Gamma_2 \,\} \cup \{\sigma = \tau{\rightarrow}t\})$,
where t is a fresh type variable
in
$S\Gamma_1 \cup S\Gamma_2 \triangleright MN : St$

$PT(\lambda x.M, A) =$
let $\Gamma \triangleright M : \tau = PT(M, A)$
in
if $x{:}\sigma \in \Gamma$ for some σ
then $\Gamma - \{x{:}\sigma\} \triangleright \lambda x.M : \sigma{\rightarrow}\tau$
else $\Gamma \triangleright \lambda x.M : s{\rightarrow}\tau$,
where s is a fresh type variable

$PT(\texttt{let } x = M \texttt{ in } N, A) =$
let $\Gamma_1 \triangleright M : \sigma = PT(M, A)$
$A' = A \cup \{x \mapsto \Gamma_1 \triangleright M : \sigma\}$
$\Gamma_2 \triangleright N : \tau = PT(N, A')$
$S = UNIFY(\{\, \alpha = \beta \mid y{:}\alpha \in \Gamma_1 \text{ and } y{:}\beta \in \Gamma_2 \,\})$
in $S\Gamma_1 \cup S\Gamma_2 \triangleright \texttt{let } x = M \texttt{ in } N : S\tau$

Figure 4: Algorithm PT to compute principal typing.