Unification and ML Type Reconstruction*

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

Harry G. Mairson‡
Department of Computer Science
Brandeis University
Waltham, Massachusetts 02254

John C. Mitchell§
Department of Computer Science
Stanford University
Stanford, California 94305

May 18, 1990

Abstract

We study the complexity of type reconstruction for a core fragment of ML with lambda abstraction, function application, and the polymorphic `let` declaration. We derive exponential upper and lower bounds on recognizing the typable core ML expressions. Our primary technical tool is unification of succinctly represented type expressions. After observing that core ML expressions, of size n, can be typed in DTIME(2^n), we exhibit two different families of programs whose principal types grow exponentially. We show how to exploit the expressiveness of the `let`-polymorphism in these constructions to derive lower bounds on deciding typability: one leads naturally to NP-hardness and the other to DTIME(2^{n^k})-hardness for each integer $k \geq 1$. Our generic simulation of any exponential time Turing Machine by ML type reconstruction may be viewed as a nonstandard way of computing with types. Our worst-case lower bounds stand in contrast to practical experience, which suggests that commonly used algorithms for type reconstruction do not slow compilation substantially.

*To appear as a chapter in **Computational Logic: Essays in Honor of Alan Robinson**, ed. J.-L. Lassez and G. Plotkin, MIT Press. Preliminary versions of the results in this paper appeared in the *Proc. 16-th ACM Symposium on Principles of Programming Languages*, pages 105–115, January 1989, and in the *Proc. 17-th ACM Symposium on Principles of Programming Languages*, pages 382–401, January 1990.

†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.

‡Supported by grants from Texas Instruments and from the Tyson Foundation.

§Supported by an NSF Presidential Young Investigator Award with matching funds from Digital Equipment Corporation, the Powell Foundation, and Xerox Corporation; and NSF grant CCR-8814921.

1 Introduction.

A convenient feature of the programming language ML [GMW79, Mil85] is the manner in which *type reconstruction*[1] is used to eliminate the need for type declarations [Mil78, DM82]. When the programmer enters an untyped expression, 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 $t \longrightarrow t$, 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. An added bonus, typical of untyped functional languages, is that code may be reused on different data types: for example, Id defined above serves as the identity function on any type t of data, i.e., it is *polymorphic*. Since the combination of ML polymorphism and type reconstruction has proven very useful in practice, the main ideas have also been adopted in other languages, such as Miranda [Tur85] and Haskell [HW88].

To simplify our analysis, we will focus on *core* ML expressions without recursion, using only lambda abstraction, function application, and `let`. Choosing a small fragment of ML makes our lower bound applicable to any extension. Given a core ML expression M, the ML *type reconstruction problem* is to find a type for M if one exists, and otherwise reject M as untypable. Our lower bound will actually apply to the simpler *recognition problem:* given a core ML expression M, determine whether or not M is typable. It is clear that any algorithm for the type reconstruction 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 outstanding difference between ML and the first order typed lambda calculus is the `let` declaration, which is crucial to ML polymorphism. For example, when we declare a function f by saying `let` $f = \ldots$, different occurrences of f within the scope of this definition 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. The facility of the ML language to reuse code (in this instance, the code $\lambda x.x$ bound to f) on different data types is realized precisely through such type polymorphism.

ML polymorphism adds succinctness to the language, whereby very long expressions in the simply typed lambda calculus can be stated equivalently by short expressions using `let`. However, a consequence is that deciding whether a core ML expression is typable becomes much more difficult than deciding the typability of `let`-free expressions. This is another example of a fundamental theme from complexity theory, namely, the use of succinctness for deriving unconditional lower bounds on natural computational problems, e.g., [Mey72, SM73].

Note that without `let`, ML type reconstruction can be done efficiently. This is type reconstruction for the simply typed lambda calculus, [CF58, Hin69, Wan87]. Using a linear time unification algorithm (e.g., [PW78]), we can compute the principal type of any `let`-free core ML expression in linear time. Even in this simple case, however, one must be

[1] Following a proposal of Albert Meyer, we use the more precise term "type reconstruction" instead of "type inference". This is because type inference is also used ambiguously to denote the process of "type derivation" via inference rules, which can be thought of as the inverse of reconstruction.

careful with the representation of type expressions. To achieve linear time, types must be represented and printed out as directed acyclic graphs, or *dags* (see [AHU]), since the string representation of a type may be exponentially longer than the expression to be typed. Dag representations are a common data structure in unification [PW78, MM82, DKM84].

For a core ML expression with `let`, we give a straightforward deterministic type reconstruction algorithm which is exponential in the number of nested `let` declarations in the expression. The algorithm is based on a type-preserving transformation of the expression into a `let`-free equivalent. However, the type of an ML expression may have doubly exponential size when represented as a string, and singly exponential size as a dag.

In studying the complexity of ML typing, we think of the meaning of ML expressions as their *types* represented by dags. A type reconstruction computation (such as the commonly used algorithm of [DM82]) proceeds in a syntax directed manner, where the types of expressions are derived via unification. Because of the *succinctness* introduced by `let`-polymorphism, the resulting unification is harder than classical first-order unification [Rob65, PW78, MM82]. In fact, as we show, the resulting unification can simulate any exponential time Turing Machine computation. This generic simulation illustrates how type reconstruction can be used (in an unconventional fashion) for computing with types.

In Section 2, we review some basic notions. We follow the exposition of [MH88, Mit90]. We observe that the inference rule for `let` is not the one found in [DM82]. This alternative exposition does not change the semantics. Its primary virtues are the explicit identification of `let` as a powerful *abbreviation* mechanism in the language, and the complete elimination of type schemes by reducing them to types. The Appendix includes a description of the algorithm and inference rules for the system in [DM82], and an equivalence theorem with our inference system.

In Section 3, we exhibit two different constructions of ML expressions whose principal types grow exponentially. In Sections 4 and 5, by carefully studying the expressiveness of the use of `let`-polymorphism in these constructions, we show how they can be used as the respective foundations for lower bounds on deciding whether an ML expression is typable. Each of the lower bound proofs uses lambda calculus programming of approximately the same sophistication as the proof of Turing completeness for untyped lambda calculus [Bar84]. The first of these constructions gives a natural encoding of truth tables, and as a consequence, we derive an NP-hardness bound on typability (see Section 4). We include this construction because of its simplicity. The second construction uses many of the techniques of the first, but leads to stronger lower bounds (see Section 5). It shows how polymorphic ML expressions mapping types to types can be composed an exponential number of times by polynomial sized terms. As a consequence, we derive unconditional exponential time lower bounds, and show that recognizing the typable core ML expressions is a problem complete in DTIME(2^{n^k}) for every integer $k \geq 1$. We conclude in Section 6.

The lower bounds contradict what appears to be a well-known "folk theorem," namely that types for ML expressions can be inferred in linear time[2]. They also stand in contrast to the perceived efficiency of the algorithm in practice.

We note that exponential lower bounds on ML type reconstruction have also been shown (independently and using altogether different methods) in [KTU90a].

[2]To the embarrassment of the third author, the incorrect "folk theorem" was put in print in [MH88]. A quadratic time bound for the problem was also claimed in [Lei83].

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 (cf. [DM82, Mil78]). In writing expressions, we use parentheses and the usual conventions of the lambda calculus. For example, MNP should be read as $((MN)P)$, and $\lambda x.MN$ read as $\lambda x.(MN)$.

In $\lambda x.N$ and $\texttt{let } x = M \texttt{ in } N$, the variable x becomes bound in N. We use the standard conventions for free and bound variables. An expression is *closed* if all variables are bound.

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. More formally, we define α-equivalence using the equalities

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

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

where $[M/x]N$ denotes the result of substituting M for free occurrences of x in N (with renaming of bound variables to avoid capture, as usual).

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

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

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

Since $\texttt{let } x = M \texttt{ in } N$ and $(\lambda x.N)M$ both reduce to $[M/x]N$, the `let`-reduction of the former and the β-reduction of the latter produce the same final value. However, they may be typed differently. For example, $\texttt{let } I = \lambda x.x \texttt{ in } II$ is typable in ML, but $(\lambda I.II)(\lambda x.x)$ is not, while both terms reduce to $(\lambda x.x)(\lambda x.x)$.

We say M' `let`-*reduces to* N', and write $M' \stackrel{let}{\twoheadrightarrow} N'$, 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]. The idea is that $\texttt{let } x = M \texttt{ in } N$ may be regarded as the *marked* redex $(\lambda x.N)^1 M$, as opposed to the unmarked redex $(\lambda x.N)M$. 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 Proposition 2.1, `let` normal forms exist and are unique.

Finally, we define the *length* of an ML expression as:

$$
\begin{array}{lcll}
|x| & = & 1 & \text{where } x \text{ is a variable} \\
|MN| & = & |M| + |N| + 1 & \\
|\lambda x.N| & = & |N| + 1 & \\
|\texttt{let } x = M \texttt{ in } N| & = & |M| + |N| + 1 &
\end{array}
$$

2.2 Types and typing assertions.

The type expressions of core ML have the abstract syntax

$$\tau \quad ::= \quad t \mid \tau \longrightarrow \tau,$$

where t may be any type variable. We use parentheses and the syntactic convention that $\longrightarrow$ associates to the right. For example, $\sigma \longrightarrow \tau \longrightarrow \varrho$ should be read as $(\sigma \longrightarrow (\tau \longrightarrow \varrho))$.

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 $\Gamma \triangleright M\colon \tau$, where M is an ML expression, τ is a type expression, and Γ is a *type assignment*: a finite set $\Gamma = \{x_1\colon \tau_1, \ldots, x_k\colon \tau_k\}$ associating at most one type with each variable x.

The assertion $\Gamma \triangleright M\colon \tau$ may be read, "the expression M has type τ in context Γ."

We say M *is typable* if there is some *provable* typing assertion $\Gamma \triangleright M\colon \tau$ about M. Typing assertions are proved using the ML *inference system* with the axioms and rules:

$$(var) \qquad \Gamma \oplus x\colon \tau_1 \triangleright x\colon \tau_1$$

$$(abs) \qquad \frac{\Gamma \oplus x\colon \tau_1 \triangleright M\colon \tau_2}{\Gamma \triangleright (\lambda x.M)\colon \tau_1 \rightarrow \tau_2}$$

$$(app) \qquad \frac{\Gamma \triangleright M\colon \tau_1 \rightarrow \tau_2, \quad \Gamma \triangleright N\colon \tau_1}{\Gamma \triangleright MN\colon \tau_2}$$

$$(\texttt{let}) \qquad \frac{\Gamma \triangleright M\colon \tau_1, \ \Gamma \triangleright [M/x]N\colon \tau}{\Gamma \triangleright \texttt{let } x = M \texttt{ in } N\colon \tau}$$

where $\Gamma \oplus x\colon \tau_1$ is the result of removing any statement about x from Γ and adding $x\colon \tau_1$.

We use this inference system in our exposition. We observe that the inference rule for `let` is not the one found in [DM82], though it does not change the semantics. Its primary virtues are the explicit identification of `let` as a powerful "abbreviation" mechanism in the language, and the complete elimination of "type schemes" by reducing them to type expressions. The Appendix includes a description of the algorithm and inference rules for the system in [DM82], and an equivalence theorem with the inference system above.

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 $S\Gamma = \{x\colon S\tau | x\colon \tau \in \Gamma\}$. A typing statement $\Gamma' \triangleright M\colon \tau'$ *is an instance of* $\Gamma \triangleright M\colon \tau$ if there exists a substitution S with $\Gamma' \supseteq S\Gamma$, and $\tau' = S\tau$.

A typing assertion $\Gamma \triangleright M\colon \tau$ is a *principal typing for* M if it is provable, and has every provable typing assertion for M as an instance. When M is closed, one can show that the principal typing will have an empty type assignment, then we say M has a *principal type*.

In order to clarify the operation of the above inference system we need one lemma. The idea is that `let` reduction (almost) preserves typings: `let` $x = M$ `in` N has the same typings as $[M/x]N$ (provided M is typable or x occurs in N). The proof of the following is straightforward, by inspection of the inference rules.

Lemma 2.2 *A core ML expression of the form* `let` $x = M$ `in` N *has precisely the same typings as the expression* `let` $x = M$ `in` $(\lambda y.N)x$*, provided* y *is not free in* N*. Furthermore,* `let`*-reduction of the latter expression to* $[M/x]((\lambda y.N)x)$ *preserves all such typings.*

Let M be a core ML expression and let M' be the result of modifying every `let` in M by adding y's and x's as in the above lemma. Then N', the unique `let` normal form of M' (see Proposition 2.1), and M have exactly the same typings. But, for `let` normal forms the above inference system is the one for the first order typed lambda calculus [CF58, Hin69]. It is well known that, for the first order typed lambda calculus provable assertions are closed under substitution and principal typings exist. This leads us to two basic properties of ML (first shown in [Mil78, DM82]).

Proposition 2.3 *If* $\Gamma' \triangleright M\colon \tau'$ *is an instance of a provable typing assertion* $\Gamma \triangleright M\colon \tau$*, then* $\Gamma' \triangleright M\colon \tau'$ *is provable.*

Proposition 2.4 *If* M *is typable, then* M *has a principal typing. If* M *is typable and closed, then* M *has a principal type.*

Since we analyze the relationship between the size of ML expressions and their respective types, these notions must be made precise. We define the *length* of a type expression τ as follows:

$$\begin{array}{lcll} |t| & = & 1 & \text{if } \tau \equiv t \text{ for some type variable } t \\ |\rho \longrightarrow \sigma| & = & 1 + |\rho| + |\sigma| & \text{if } \tau \equiv \rho \longrightarrow \sigma \text{ is a compound type} \end{array}$$

When types are represented as dags, we define the size of the dag to be the number of its nodes. Internal nodes have outdegree two and are labelled with $\longrightarrow$. Leaves are labelled with distinct type variables (i.e., we eliminate common subexpressions that are type variables).

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 $R = T \circ S$ ($\circ$ denotes function composition).

Proposition 2.5 (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 use of unification to combine typing statements about subexpressions. For example, if $M\colon \sigma \longrightarrow \tau$ and $N\colon \varrho$, then we must unify σ and ϱ in order to type the application MN. If σ and τ share type variables, then the type of MN will be τ subject to the further constraints induced by the unification of σ and ϱ.

While most implementations of unification have slightly higher asymptotic running time, unification can be done in linear time e.g., [PW78]. To perform unification efficiently, it is common to represent the expressions to be unified (in our case, type expressions) as directed acyclic graphs, [AHU]. A directed acyclic graph (*dag*) representation is like the parse tree of an expression, with each node labelled by an operator or operand (in our case, $\longrightarrow$ or a type variable). 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.

The time required for unification is *linear in the size of the dags* representing the expressions to be unified. As we shall see, closed core ML expressions of size m can have principal types of dag size 2^m. A consequence of this succinct representation is that the unification of dags represented by ML expressions can be very costly.

3 Upper bounds on type reconstruction.

3.1 Type size and reconstruction 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

$$\langle M_1, \ldots, M_k \rangle ::= \ \lambda z.zM_1 \ldots M_k$$

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

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

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 $(\sigma_1 \longrightarrow \ldots \longrightarrow \sigma_k \longrightarrow t) \longrightarrow t$ with t not occurring in any σ_i.

Let I and K be the familiar combinators $\lambda x.x$ and $\lambda x.\lambda y.x$. Observe that when an ML formula $\lambda w.Kw\langle \phi_1, \ldots, \phi_n \rangle$ is typable and no ϕ_i contains a free occurence of w, it has the same principal type as the I combinator. Such a formula might be untypable if the type constraints introduced by the ϕ_i cannot be satisfied. In our lower bounds analysis, we will use ML expressions to construct dags describing types via unification; the above construct allows a transparent means of introducing constraints on types of subexpressions.

Example 3.6 *The closed expression*

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

has principal type $t \longrightarrow (t \times t)$. 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 3.6 to any typable expression M (e.g., $P\,(P\,\ldots(P\,M)\ldots))$, we can prove:

Proposition 3.7 *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 3.8 *Given a* `let`*-free expression M of length n (with all bound variables distinct), 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 at most $2^{O(n)}$ 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 – recall that all its bound variables are distinct; (2) for each node of the parse tree pick a type variable denoting the principal type of the lambda term rooted at that node; (3) write constraints for each node as equations among type expressions, where there are three kinds of equations, 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; a correctness proof for essentially this algorithm appears in [Wan87]. 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 Damas-Milner ML typing algorithm on `let` expressions (algorithm PT in the Appendix). 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.) Because of variable renaming, the number of type variables involved in the principal type of an expression may be exponential.

The following example illustrating this exponential growth is due to Mitchell Wand and, independently, to Peter Buneman.

Example 3.9 *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. Unlike the expression $(\lambda x.\langle x, x\rangle)M$ in Example 3.6, 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*

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

with n nested declarations. The principal types of the W_n 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 3.9 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. In the following example, we construct expressions such that the principal types have doubly-exponential size when written out as a string, and the dag representations of *any* typing must have at least exponential size.

Example 3.10 *Recall that the expression $P ::= \lambda x.\langle x, x\rangle$ from Example 3.6 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 a factor of 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. The dag must contain an exponential-length path, thus any substitution instance of the principal type also has exponential dag size. Such an expression V_n is defined as follows:*

$$
\begin{array}{l}
V_n ::= \\
\quad \texttt{let}\ x_0 = \lambda x.\langle x, x\rangle\ \texttt{in} \\
\quad\quad \texttt{let}\ x_1 = \lambda y.x_0(x_0 y)\ \texttt{in} \\
\quad\quad\quad \ldots \\
\quad\quad\quad \texttt{let}\ x_n = \lambda y.x_{n-1}(x_{n-1} y)\ \texttt{in}\ x_n(\lambda z.z)
\end{array}
$$

To write the principal type of this expression simply, let us use the notation $\tau^{[k]}$ for the n-ary product defined inductively by $\tau^{[1]} ::= \tau$ and $\tau^{[k+1]} ::= (\tau^{[k]}) \times (\tau^{[k]})$. Observe that $\tau^{[n]}$ has $2^{\Omega(n)}$ symbols. By examining the expression V_n and tracing the behavior of the ML *typing algorithm, we see that* $x_0 \equiv P$ *has principal type* $t \longrightarrow t^{[2]}$*, and for each* $k > 0$ *the principal type of* x_k *has type* $t \longrightarrow t^{[2^k+1]}$*. Consequently, the principal type of the entire expression* V_n *is* $(t \longrightarrow t)^{[2^n+1]}$*, which has length* $2^{2^{\Omega(n)}}$*. Since a dag representation can reduce this by at most one exponential, the dag size of the principal type is* $2^{\Omega(n)}$*.*

Proposition 3.11 *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 $n \geq 1$, we may construct an expression W_n as in Example 3.9 whose principal type has exponentially many type variables, and an expression V_n as in Example 3.10 satisfying the remaining conditions. The expression $\langle W_n, V_n \rangle$ proves the proposition. ∎

3.3 Type reconstruction with `let`.

One method to type a core ML expression is simply to reduce to `let` normal form (after some simple rewriting), and to then use the linear time algorithm of Proposition 3.8. The primary reason this method works properly is that types are preserved by `let` reduction, as described in Lemma 2.2. From this lemma and Proposition 2.4 we have:

Lemma 3.12 *A core ML expression of the form* $\texttt{let}\ x = M\ \texttt{in}\ N$ *has the same principal typing as the expression* $\texttt{let}\ x = M\ \texttt{in}\ (\lambda y.N)x$*, provided* y *is not free in* N*. Furthermore,* `let`*-reduction of the latter expression to* $[M/x]((\lambda y.N)x)$ *preserves the principal type.*

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.

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

In the common special case that M is `let`-free, the `let`-depth of $\texttt{let}\ x = M\ \texttt{in}\ N$ is $1 + \ell d(N)$.

Lemma 3.13 *Let* E' *be the* `let`*-normal form of an ML expression* E*. Then* $|E'| \leq |E|^{\ell d(E)}$*.*

Proof. By Proposition 2.1, the `let`-normal form is always defined and is unique. We proceed by 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.

$$\begin{array}{rcll} |(MN)'| & = & |M'N'| & \\ & = & |M'| + |N'| + 1 & \\ & \leq & |M|^{\ell d(M)} + |N|^{\ell d(N)} + 1 & \text{by inductive hypothesis} \\ & \leq & |M|^{\max\{\ell d(M), \ell d(N)\}} + |N|^{\max\{\ell d(M), \ell d(N)\}} + 1 & \\ & \leq & (|M| + |N| + 1)^{\max\{\ell d(M), \ell d(N)\}} & \\ & \leq & |MN|^{\ell d(MN)}. & \end{array}$$

The case of lambda abstraction is similar. The remaining case is $\texttt{let}\ x = M\ \texttt{in}\ N$, where it becomes clear how the definition of let-depth was made to facilitate the induction.

$$
\begin{array}{rcll}
|(\texttt{let}\ x = M\ \texttt{in}\ N)'| & = & |(\texttt{let}\ x = M'\ \texttt{in}\ N')'| & \\
 & = & |[M'/x]N'| & \\
 & \leq & |M'| \cdot |N'| & \\
 & \leq & |M|^{\ell d(M)} \cdot |N|^{\ell d(N)} & \text{by inductive hypothesis} \\
 & \leq & (|M| + |N| + 1)^{\ell d(M) + \ell d(N)} & \\
 & \leq & |\texttt{let}\ x = M\ \texttt{in}\ N|^{\ell d(\texttt{let}\ x=M\ \texttt{in}\ N)} &
\end{array}
$$

∎

By these lemmas (using simple rewriting, then the finiteness of developments and then first-order unification) we have that:

Theorem 3.14 *There is an algorithm which decides whether any given core ML expression M has a provable typing in time $O(|M|^\ell)$, where ℓ is the* let*-depth of M. When M is typable, this algorithm yields a dag representing the principal typing.*

Corollary 3.15 *For fixed* let*-depth ℓ, there are polynomial time algorithms to determine if an ML expression of* let*-depth ℓ is typable, as well as to compute the principal type.*

Corollary 3.16 *([Bar84, Ch. 11, Exercise 11.5.8(i)]) If M is a core ML expression of length n, then the* let*-normal form of M is of length at most $2^{O(n)}$. The principal typing of M has length at most doubly-exponential in n, and dag size $2^{O(n)}$.*

Proof. Redo the induction of 3.13, changing the induction hypothesis accordingly. ∎

4 An NP-hardness bound for typability.

We have seen that there exist exponential time algorithms to decide if an ML expression is typable, and to compute the principal type if the expression is indeed typable. The algorithms run in polynomial time in the case of fixed let-depth. We now examine the typability question in more detail, using the construction W_n of Example 3.9 to derive a simple NP-hardness bound on deciding if an ML expression is typable.

It is possible to use let-polymorphism in the style of Example 3.9 to simulate rows and columns of truth assignments in a truth table. This simulation allows a reduction from the satisfiability problem: given a propositional formula Φ in conjunctive normal form, we construct an ML expression E_Φ of length polynomial in the length of Φ, where Φ is satisfiable iff E_Φ is *not* typable. We write $\Phi \equiv C_1 \wedge C_2 \wedge \cdots \wedge C_m$, where each clause C_i is the disjunction of a set of literals chosen from the set $\{v_1, \overline{v_1}, v_2, \overline{v_2}, \ldots, v_n, \overline{v_n}\}$.

Let the Boolean variables $v_1, v_2, \ldots v_n$ appearing in Φ be arranged in a truth table. The column of the truth table associated with v_n looks like $T, F, T, F, \ldots$, the column associated with v_{n-1} looks like $T, T, F, F, T, T, F, F, \ldots$, and so on. In general, the column associated with v_{n-j} consists of a block of 2^j Ts followed by 2^j Fs, this block repeated 2^{n-j-1} times. This kind of regular pattern is ideally simulated by the construction in Example 3.9. We simulate the $(n-j)$th column of the truth table (associated with v_{n-j}) using the following expression:

Example 4.17

v_{n-j} =

```
let x0 = T_{n-j} in
    let x1 = ⟨x0, x0⟩ in
        ...
        let xj = ⟨x_{j-1}, x_{j-1}⟩ in
            let y0 = F_{n-j} in
                let y1 = ⟨y0, y0⟩ in
                    ...
                    let yj = ⟨y_{j-1}, y_{j-1}⟩ in
                        let z0 = ⟨xj, yj⟩ in
                            let z1 = ⟨z0, z0⟩ in
                                ...
                                let z_{n-j-1} = ⟨z_{n-j-2}, z_{n-j-2}⟩ in
                                    z_{n-j-1}
```

If we interpret the pairing operator $\langle \cdot, \cdot \rangle$ as constructing a binary tree from two arguments comprising the left and right subtrees, then the above expression `let`-reduces to an exponential-sized complete binary tree. The *type* of v_j has a similar tree structure: the pairing operator has the type of a dag, with leaves in the dag to be unified with the types of the left and right subtrees. The leaves of the tree v_j are either T_j or F_j: when enumerated from left to right, the leaves encode a column of the truth table. The type of T_j will encode which clauses of Φ are forced to be true by choosing the truth valuation of v_j to be *true*, and the type of F_j will encode which clauses of Φ are forced to be true by choosing the truth valuation of v_j to be *false*.

When the types of the binary trees v_j and $v_{j'}$ are unified, the truth information in the leaves is combined pairwise: the kth leaf in the unification of the trees indicates, for each clause C_i of Φ, whether the truth assignments encoded in the kth leaves of v_j and $v_{j'}$ force C_i to be true. Iterating this unification, we define

$$\textit{not-satisfiable?} \quad = \quad \lambda x.\lambda z.K\ z\ \langle Eq(x, v_1), Eq(x, v_2), \ldots, Eq(x, v_n), Eq(x, \textit{sat-test})\rangle$$

This definition introduces another complete binary tree *sat-test* which tests satisfiability, and a special binary combinator Eq which forces its two arguments to have the same type. Their use above forces all of the trees v_j to be unified, so that the type of each leaf in the tree x encodes a *row* of the truth table, together with a local satisfiability tester for that row. The encoding of the row indicates which clauses of Φ are true under the truth assignment given by that row. If the encoding satisfies Φ, then the local satisfiability tester forces a mistyping.

To construct the required ML expressions, we use the Eq combinator, which enforces via unification the type equality of its two arguments:

$$Eq = \lambda x.\lambda y.\ K\ x\ \lambda z.\ K\,(z\,x)\,(z\,y) :\ a \longrightarrow a \longrightarrow a.$$

For clarity, we will occasionally indicate (using a :) the principal type of an expression, in this case : $a \longrightarrow a \longrightarrow a$. Observe that as the lambda-bound variable z is applied to both x and y in the definition of Eq, then if u and v are ML expressions, the expression $Eq\ u\ v$ (which we will usually write as $Eq(u, v)$) can only be typed if u and v have the same type.

Let the clauses in the proposition Φ be uniquely labelled $1, 2, \ldots, m$. Suppose the literal v_j appears in clauses labelled $a_{j,1}, a_{j,2}, \ldots a_{j,P(j)}$, and literal $\overline{v_j}$ appears in clauses labelled $a_{\overline{j},1}, a_{\overline{j},2}, \ldots a_{\overline{j},N(j)}$. We then define ML terms T_j and F_j, $1 \leq j \leq n$, as:

$$\begin{aligned} T_j &= \lambda x_1.\lambda y_1.\lambda x_2.\lambda y_2.\cdots\lambda x_m.\lambda y_m.\lambda z.K\ z \\ &\qquad \langle Eq(x_{j,1}, y_{j,1}), Eq(x_{j,2}, y_{j,2}), \ldots, Eq(x_{j,P(j)}, y_{j,P(j)})\rangle \\ F_j &= \lambda x_1.\lambda y_1.\lambda x_2.\lambda y_2.\cdots\lambda x_m.\lambda y_m.\lambda z.K\ z \\ &\qquad \langle Eq(x_{\overline{j},1}, y_{\overline{j},1}), Eq(x_{\overline{j},2}, y_{\overline{j},2}), \ldots, Eq(x_{\overline{j},N(j)}, y_{\overline{j},N(j)})\rangle \end{aligned}$$

When the type of T_j is described as a dag, it appears as a "comb" with a long right spine, and $2m$ "children" hanging off the left hand side of the spine. These children correspond to the types of the x_i and y_i, $1 \leq i \leq m$; when the types of x_i and y_i are equal, it means precisely that setting v_j to true forces the ith clause of Φ to be true. Observe that the constraints in the brackets are "transparent" due to the use of the K combinator, in that they affect the types of the x_ℓ and y_ℓ, but do not otherwise appear explicitly as part of the type structure of the expression.

We now define a term *sat* whose type is like that of the T_j and F_j, except that it encodes the effect of a truth valuation which would satisfy *all* the clauses in Φ:

$$\begin{aligned} sat &= \lambda x_1.\lambda y_1.\lambda x_2.\lambda y_2.\cdots\lambda x_m.\lambda y_m.\lambda z.K\ z \\ &\qquad \langle Eq(x_1, y_1), Eq(x_2, y_2), \ldots Eq(x_m, y_m)\rangle \end{aligned}$$

The relation between a satisfying truth valuation of Φ and *sat* is expressed by the following proposition:

Lemma 4.18 *Let $\mathcal{R} = \{X_1, X_2, \ldots, X_n\}$ be a set of ML expressions such that each $X_j \in \{T_j, F_j\}$. Let R be the type derived from unifying the types of the $X_j \in \mathcal{R}$ together. Let $\nu: \{v_j \mid 1 \leq j \leq n\} \longrightarrow \{\mathit{true}, \mathit{false}\}$ be the natural truth assignment induced by $\mathcal{R}$, where $\nu(v_j) = \mathit{true}$ if $X_j = T_j$, and $\nu(v_j) = \mathit{false}$ if $X_j = F_j$. Then ν satisfies Φ iff R is the principal type of sat.*

Proof. If $\mathcal{R}$ interpreted as a truth assignment satisfies Φ, then every clause with label i must be satisfied by some literal represented by X_j. The types associated with the variables x_i and y_i appearing in X_j must then be equal, and this forces the corresponding part of R to have the same structure. Since this occurs for each clause, R is the principal type of *sat*. The argument is similar when $\mathcal{R}$, interpreted as a truth assignment, does not satisfy Φ: in this case the principal type of *sat* is a proper instance of R. ∎

Since our goal is to show that the expression *not-satisfiable?* is typable iff Φ is not satisfiable, we introduce a new ML term called a *tester*: its function is to cause a mistyping if its type is unified with a type encoding a truth assignment satisfying Φ.

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

Lemma 4.19 *The ML expression* (Eq *sat tester*) *cannot be typed.*

Proof. The principal types of *sat* and *tester* are very similar, except for the constraints on the parts of the types associated with the x_i and y_i. The Eq combinator forces the types of *sat* and *tester* to be unified: *sat* forces the types of each x_i and y_i to be identical, while *tester* forces the types of each y_i and x_{i+1} to be identical. A consequence of this chain of equalities is that the types of x_1 and y_m are forced to unify and thus be equal. But if the types of x_1 and y_m are equal, then the subexpression $x_1 y_m$ in *tester* cannot be typed. ∎

Imitating the structure of the complete binary trees v_j, we use `let`-polymorphism to define the complete binary tree *sat-test*, where a copy of *tester* appears at each leaf:

$$
\begin{array}{l}
\textit{sat-test} = \\
\quad \texttt{let } t_1 = \langle \textit{tester}, \textit{tester} \rangle \texttt{ in} \\
\quad\quad \texttt{let } t_2 = \langle t_1, t_1 \rangle \texttt{ in} \\
\quad\quad\quad \ldots \\
\quad\quad\quad \texttt{let } t_n = \langle t_{n-1}, t_{n-1} \rangle \texttt{ in } t_n
\end{array}
$$

Recall that the ML expression *(not-satisfiable?)* has been defined above using the expressions v_j, Eq and *sat-test*.

Lemma 4.20 *The expression* (*not-satisfiable?*) *can be typed iff* Φ *is not satisfiable.*

Proof. When Φ is satisfiable, some row of the truth table satisfies Φ. By Lemma 4.18, some leaf in the binary tree defined by the unification of the types of the trees v_j has the type of *sat*. Lemma 4.19 then shows that the further unification of this leaf with the type of *tester* forces a mistyping. When Φ is not satisfiable, each row of the truth table fails to satisfy Φ. The corresponding leaf derived via unification does not have the type of *sat*, since some clause C_ℓ is not satisfied, and the types of x_ℓ and y_ℓ are not constrained to be equal. This lack of constraint "breaks the chain" of type equalities, so further unification with the type of *tester* does not cause a mistyping. ∎

Theorem 4.21 *Deciding if an ML expression is typable is NP-hard.*

We observe that by elaborating some of the above constructions, Theorem 4.21 can be strengthened to derive a PSPACE-hardness lower bound [KM89]. This stronger bound involves the introduction of Boolean logic gates in the place of the pairing operator $\langle \cdot, \cdot \rangle$, so that the subtrees act like inputs in a Boolean circuit. The virtue of the NP-hardness proof is its brevity and directness; rather than detail the PSPACE-hardness bound, we proceed to stronger results using a different approach.

5 An exponential lower bound on typability.

In contrast to the reduction in the previous section, we now present a *generic* reduction: given any deterministic one-tape Turing machine M with input x running in $O(2^{|x|^k})$ time, $k \geq 1$, we show how to construct an ML formula $\Psi_{M,x}$, such that M accepts x iff $\Psi_{M,x}$ is typable. In this reduction, the length of $\Psi_{M,x}$ is polynomial in the length of the description of M and x, and exponential in k. Since every language L in DTIME(2^{n^k}) has a deterministic Turing Machine M_L which can decide if $x \in L$ for input x in $O(2^{|x|^k})$ time, this reduction shows that the difficulty of deciding typability of ML expressions is complete for each complexity class DTIME(2^{n^k}), $k \geq 1$. We refer to these languages collectively as the class DEXPTIME.

The simple intuition providing the foundation of the generic reduction and associated lower bounds is motivated by the construction of V_n in Example 3.10. The intuition is the following: note that the function x_n in the example is equivalent to the lambda term $\lambda y.x_0^{2^n} y$, namely, a function which applies the x_0 function an exponential number of times to its argument. If y was a piece of Turing Machine tape, and x_0 was a function which added a tape square to the tape, x_n would be a good function for constructing exponential-sized Turing Machine IDs. If y was a Turing Machine ID, and x_0 was its transition function δ, x_n would be a good way to "turn the transition crank" and apply δ an exponential number of times to the initial machine ID. Of course, there are many technical details to work out, but the inspiration is simply that the "exp" in "exponential function composition" is the same "exp" as in "DEXPTIME." It is uniquely the expressive power of ML polymorphism to succinctly express function composition which permits the polynomial-time (and, in fact, logspace) reduction.

In our proof, the technical mechanics simulating the transition function δ of the Turing Machine are realized purely through terms in the lambda calculus *without* the use of the polymorphic `let` construct. The transition function can be represented in a straightforward manner by a Boolean circuit, where the inputs are variables q_i set to *true* iff the machine is in state i, and variables z and o indicate whether the tape head is reading a 0 or a 1. The output of the circuit indicates the new state, what is written on the tape cell, and the head direction. All of this circuitry is realized by lambda terms, using the Boolean gadgets defined in [DKM84]. We add a Boolean "fanout" gate to this logical menagerie in the interest of facilitating our proof.

We present the proof in "bottom up" form, showing first how to encode tape symbols, the relevant Boolean logic, Turing machine state encoding, tape encoding, proceeding piece by piece to build up the entire simulation. It may come as a shock to some more practical functional programming language enthusiasts that this rather arcane lower bound *is just a computer program*, where we are interested in the *type* produced by the program instead of the *value*. The generic reduction is just a compiler: namely, how to compile Turing Machines into ML types. Since our "object code" is ML, we have endeavored to follow the gospel of [AS85] wherever possible, using modularization and data abstraction to make the program and proof more understandable.

5.1 Pairing and projection.

$$\begin{aligned} \mathit{pair} \quad = \quad & \lambda x.\lambda y.\lambda x'.\lambda y'.\lambda z.K\ z\ \langle Eq(x', x),\ Eq(y', y)\rangle : \\ & a \longrightarrow b \longrightarrow a \longrightarrow b \longrightarrow c \longrightarrow c \end{aligned}$$

The definition of *pair* introduces the type equivalent of the Lisp `cons`, and allows the use of types to build data structures. Instead of $(\mathit{pair}\ x\ y)$ we usually write $[x; y]$. When *pair* is applied to two terms x and y, the term $[x; y]$ has type $a \longrightarrow b \longrightarrow c \longrightarrow c$, where a is the type of x and b is the type of y. If u and v are ML lambda-bound variables and we need to type the function application $[x; y]\ u\, v$, then the types of x and u must be the same, as must be the types of y and v.

5.2 Boolean values: *true* and *false*.

$$
\begin{aligned}
\mathit{true} &= \lambda x.\lambda y.\lambda z.K\ z\ Eq(x,y): \qquad a \longrightarrow a \longrightarrow b \longrightarrow b \\
\mathit{false} &= \lambda x.\lambda y.\lambda z.z: \qquad a \longrightarrow b \longrightarrow c \longrightarrow c
\end{aligned}
$$

The types of *true* and *false* are virtually identical. If we regard them as functions, the only difference is that the first two (curried) arguments of *true* must be of the same type. If *true* is applied to two arguments whose types cannot be unified, for example I and Eq, then a mistyping occurs; on the other hand, *false* $I\ Eq$ can be typed.

5.3 Zero and One (Tape Symbols).

$$
\begin{aligned}
\mathit{zero} &= [\mathit{true};\mathit{false}] \\
&= \lambda x.\lambda y.\lambda z.K\ z\ \langle Eq(x,\mathit{true}), Eq(y,\mathit{false})\rangle \\
\mathit{one} &= [\mathit{false};\mathit{true}] \\
&= \lambda x.\lambda y.\lambda z.K\ z\ \langle Eq(x,\mathit{false}), Eq(y,\mathit{true})\rangle
\end{aligned}
$$

Now we define combinators that serve as predicates telling if a cell holds a zero or a one:

$$
\begin{aligned}
\mathit{zero?} &= \lambda \mathit{cell}.\lambda x.\lambda y.\lambda z.Kz\ \lambda p.\langle \mathit{cell}\ p,\ \ p\ x\ y\rangle \\
\mathit{one?} &= \lambda \mathit{cell}.\lambda x.\lambda y.\lambda z.Kz\ \lambda p.\lambda q.\langle \mathit{cell}\ p\, q,\ \ q\ x\ y\rangle
\end{aligned}
$$

Observe in the code for *zero?* that *cell* p causes the type of p to unify with the type of the "first" component in the cell, and then $p\ x\ y$ "loads" the right "type bindings" for x and y in the "answer" $\lambda x.\lambda y.\lambda z.z$, possibly unifying the types of x and y if p encodes *true*.

The definition of *zero?* also demonstrates a general style for using ML to compute with types. Note first the declarations of "inputs" (*cell*) and "outputs" (x, y). The $\lambda z.Kz$ marks the end of the inputs and outputs; next come the "local declarations," of which we have only one, for p. In the brackets, we have the "body" of the procedure. It is intuitively useful for us to think of the instructions in the body being executed from top to bottom, even if they represent a set of constraints which are being realized "simultaneously."

The importance of this encoding for zero and one is that we simulate the Boolean circuitry in the finite state control of the Turing Machine using only the *monotone* functions *and* and *or*. By encoding zero and one as these pairs of Boolean values, we do not need to simulate negation.

5.4 Monotone Boolean logic.

We implement the monotone Boolean operators *and* and *or* using gadgets similar to those introduced in [DKM84].

$$
\begin{aligned}
&\mathrm{and} = \\
&\quad \lambda in_1.\lambda in_2.\lambda u.\lambda v.\lambda z.Kz \\
&\qquad \lambda x_1.\lambda y_1.\lambda x_2.\lambda y_2.\lambda w. \\
&\qquad\quad \langle in_1\, x_1\, y_1,\ \ in_2\, x_2\, y_2, \\
&\qquad\quad\ \ x_1 u,\ y_1 w,\ x_2 w,\ y_2 v\rangle
\end{aligned}
$$

Observe that if $u{:}\,a$, $v{:}\,b$, and $w{:}\,c$, then the subterms $x_1 u,\ y_1 w,\ x_2 w,\ y_2 v$ get typed as

$$x_1^{a\longrightarrow f} u^a \qquad y_1^{c\longrightarrow g} w^c \qquad x_2^{c\longrightarrow h} w^c \qquad y_2^{b\longrightarrow k} v^b.$$

If the type of x_1 equals the type of y_1, then $a \longrightarrow f = c \longrightarrow g$ and $a = c$. If the type of x_2 equals the type of y_2, similarly $b = c$, and $a = b$ follows—namely, that the "output" variables u and v are forced into having the same type.

The definition of *and* also demonstrates a general style for using ML to compute with types. Note first the declarations of "inputs" in_1 and in_2 and "outputs" u and v. The $\lambda z.Kz$ marks the end of the inputs and outputs; next comes the "local declarations" of x_1, y_1, x_2, y_2, and w. In the brackets, we have the "body" of the procedure. In the body, the contents of in_1 and in_2 are "loaded" into the local variables, and the subsequent constraints force unification with the variables describing the output.

Now for the disjunction:

$$\begin{aligned}
&or = \\
&\quad \lambda in_1.\lambda in_2.\lambda u.\lambda v.\lambda z.Kz \\
&\qquad \lambda x_1.\lambda y_1.\lambda x_2.\lambda y_2. \\
&\qquad\quad \langle in_1\, x_1\, y_1,\ in_2\, x_2\, y_2, \\
&\qquad\quad\ \ x_1 u,\ y_1 v,\ x_2 u,\ y_2 v\rangle
\end{aligned}$$

In typing this term, we have the constraints

$$x_1^{a\longrightarrow f} u^a \qquad y_1^{b\longrightarrow g} v^b \qquad x_2^{a\longrightarrow h} u^a \qquad y_2^{b\longrightarrow k} v^b.$$

If the type of x_i equals the type of y_i for $i = 1$ or $i = 2$, then $a = b$, and the type of u equals the type of v.

Before proceeding further, we add yet another gadget to implement *multiple fanout* of Boolean values, indicating why such an addition is necessary.

Example 5.22 *Note that however strong the temptation may be, the above logic gates cannot be used in a "free" functional style if the simulation of Boolean logic is to be faithful. For example, we find (rather oddly) that (if $\sim$ stands for "has the same type as"):*

$$(\lambda p.\lambda q.\lambda r.[\mathit{or}\; p\, q; \mathit{or}\; q\, r])\ \mathit{true}\ \mathit{false}\ \mathit{false} \sim [\mathit{true}; \mathit{true}]$$

when we would have expected the right hand side to be $[\mathrm{true}; \mathrm{false}]$. *What happened? Imagine we have for $1 \leq i \leq 3$ pairs of lambda-bound variables (x_i, y_i), where the type of x_1 and y_1 are constrained to be identical in simulation of our encoding of the Boolean "true." We let (u_j, v_j), $1 \leq j \leq 2$ encode the Boolean* or *of the first two and last two pairs. The encoding of the* or *operator enforces the following constraints:*

$$\begin{array}{cccc}
x_1^{a\longrightarrow f} u_1^a & y_1^{b\longrightarrow g} v_1^b & x_2^{a\longrightarrow h} u_1^a & y_2^{b\longrightarrow k} v_1^b \\
x_2^{a\longrightarrow h} u_2^a & y_2^{b\longrightarrow k} v_2^b & x_3^{a\longrightarrow \ell} u_2^a & y_3^{b\longrightarrow m} v_2^b
\end{array}$$

Note that since x_2 is a lambda-bound variable which is applied to the lambda-bound variables u_1 and u_2, these variables are forced to have the same type. By symmetry, v_1 and v_2 are also constrained to have the same type. Thus when the p has the type of true, *the types of u_1 and v_1 are forced to be equal; unfortunately, this compels u_2 and v_2 to have the same type as well.*

What has been ignored in the Boolean simulation is that the second input q has multiple fanout: if we introduced constraints by typing the terms $\{x_2 u_1 u_2,\ y_2 v_1 v_2\}$ instead of typing $\{x_2 u_1,\ x_2 u_2,\ y_2 v_1,\ y_2 v_2\}$, then everything works out properly:

$$\begin{array}{ll} x_1^{a \longrightarrow f} u_1^a & y_1^{b \longrightarrow g} v_1^b \\ x_2^{a \longrightarrow c \longrightarrow h} u_1^a u_2^c & y_2^{b \longrightarrow d \longrightarrow k} v_1^b v_2^d \\ x_3^{c \longrightarrow \ell} u_2^c & y_3^{d \longrightarrow m} v_2^d \end{array}$$

If the types of x_2 and y_2 are equal in this example, we get $a \longrightarrow c \longrightarrow h = b \longrightarrow d \longrightarrow k$, so $a = b$ and $c = d$—both outputs are true. But if only the types of x_1 and y_1 are equal, we derive $a \longrightarrow f = b \longrightarrow g$, hence $a = b$, but we cannot derive $c = d$, the latter equality necessary to make the second output true.

This example motivates the introduction of another gadget—not to do Boolean logic, but fanout. We observe that as long as we use the fanout gate to ensure that no input is used in two different Boolean calculations, the simulation will be faithful.

$$\begin{array}{l} \mathit{fanout} = \\ \quad \lambda in.\lambda out_1.\lambda out_2.\lambda z.K z \\ \qquad \lambda u.\lambda v.\lambda x_1.\lambda y_1.\lambda x_2.\lambda y_2. \\ \qquad\quad \langle in\ u\ v, \\ \qquad\quad\ out_1\ x_1\, y_1,\ Eq(out_1, \mathsf{false}), \\ \qquad\quad\ out_2\ x_2\, y_2,\ Eq(out_2, \mathsf{false}), \\ \qquad\quad\ u\ x_1\, x_2,\ \ v\ y_1\, y_2 \rangle \end{array}$$

Viewed as a dag in the style of [DKM84] (see Figure 1), the fanout gate is just an upside-down *or* gate. Do not be misled by the Eq above: its use only constrains the types of the out_1 (and similarly, out_2) to be of the form $a \longrightarrow b \longrightarrow c \longrightarrow c$ ("false until proven true"); further constraints may force $a = b$. Figure 1 also shows the type of *fanout* as a type dag. Observe that when u and v are unified (the result of applying *fanout* to *true*), propagation forces x_i to unify with y_i for $i = 1, 2$.

By using *fanout*, we can also replicate the types of lambda terms $\lambda x_1.\lambda x_2.\cdots\lambda x_k.\lambda z.z$ where the x_i have Boolean types associated with them:

$$\begin{array}{l} copy_k = \\ \quad \lambda in.\lambda out_1.\lambda out_2.\lambda z.K z \\ \qquad \lambda u_1.\lambda u_2.\cdots\lambda u_k. \\ \qquad \lambda v_1.\lambda v_2.\cdots\lambda v_k. \\ \qquad \lambda w_1.\lambda w_2.\cdots\lambda w_k. \\ \qquad\quad \langle in\ u_1\, u_2\ \cdots\ u_k, \\ \qquad\quad \mathit{fanout}\ u_1\, v_1\, w_1,\ \mathit{fanout}\ u_2\, v_2\, w_2,\ \ldots,\ \mathit{fanout}\ u_k\, v_k\, w_k, \\ \qquad\quad Eq(out_1,\ \lambda x_1.\lambda x_2.\cdots\lambda x_k.\lambda y.y), \\ \qquad\quad Eq(out_2,\ \lambda x_1.\lambda x_2.\cdots\lambda x_k.\lambda y.y), \\ \qquad\quad out_1\ v_1\, v_2\ \cdots\ v_k,\ out_2\ w_1\, w_2\ \cdots\ w_k \rangle \end{array}$$

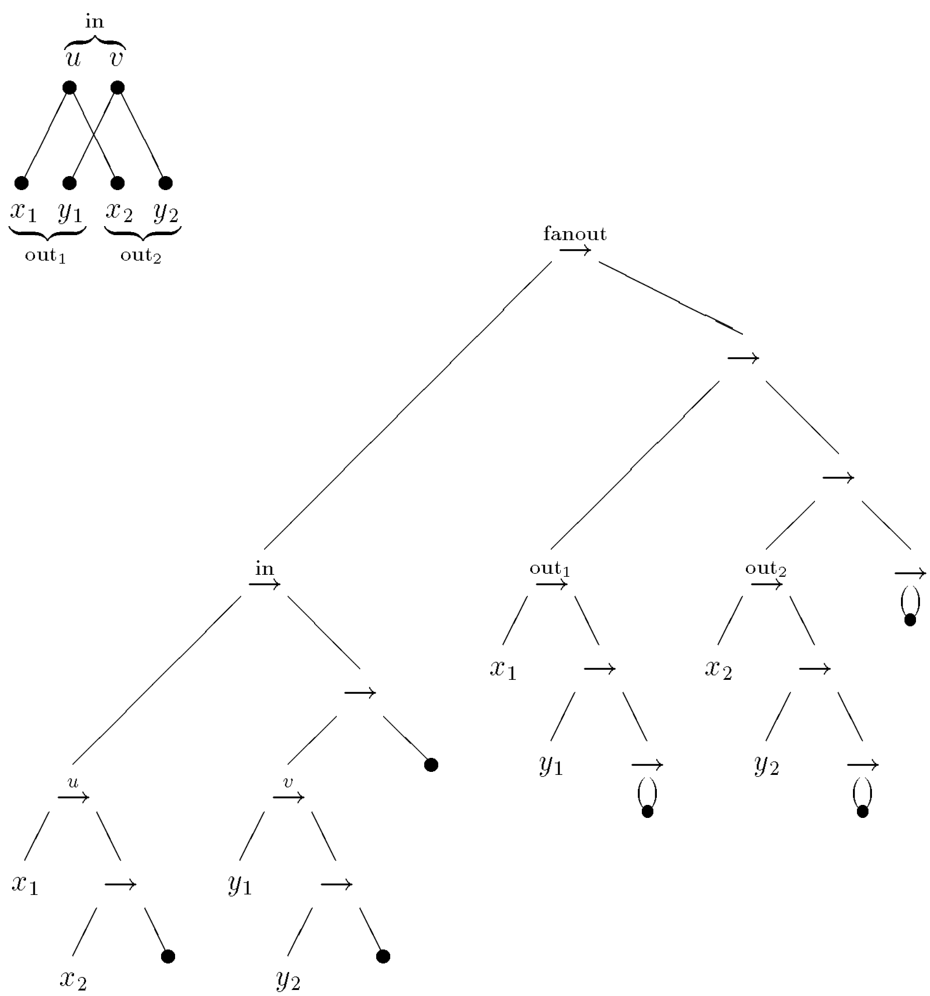

Figure 1: Implementing *fanout* as (a) a dag; (b) a type.

This definition can be used to copy tape symbols:

$$\textit{copy-cell} \quad = \quad \textit{copy}_2$$

In addition, we can use the definition of $copy_k$ to construct $j > 1$ copies of some type structure:

$$\begin{array}{l}
copy_{k,j} = \\
\quad \lambda in.\lambda out_1.\lambda out_2.\cdots\lambda out_j.\lambda z.Kz \\
\quad\quad \lambda u_1.\lambda u_2.\cdots\lambda u_j. \\
\quad\quad\quad \langle copy_k\ \ in\, u_1\, out_1, \\
\quad\quad\quad copy_k\ \ u_1\, u_2\, out_2, \\
\quad\quad\quad copy_k\ \ u_2\, u_3\, out_3, \\
\quad\quad\quad\quad \cdots \\
\quad\quad\quad copy_k\ \ u_{j-1}\, u_j\, out_j\rangle
\end{array}$$

Notice that copying or fanning-out a type tends to "corrupt" it via unification, so that using it again as an input can cause problems with the simulation of the logic. To avoid this complication, we use the "temporary" types of u_i, so that $copy_k\ u_i\, u_{i+1}\, out_{i+1}$ uses u_i to copy the type structure into out_{i+1} as well as u_{i+1}; the latter uncorrupted type is then used to continue copying.

5.5 Machine states. Testing for acceptance or rejection.

Now we commence in earnest the coding of a Turing Machine. Let its states be

$$Q = \{q_1, q_2, \ldots, q_n\}$$

where q_1 is the initial state, and the accepting and rejecting states are (respectively):

$$\begin{array}{rcl}
A & = & \{q_{\ell+1}, q_{\ell+2}, \ldots, q_m\} \\
R & = & \{q_{m+1}, q_{m+2}, \ldots, q_n\}
\end{array}$$

We now code up the ML simulation of the initial state, and how states can be replicated:

$$\begin{array}{l}
\textit{initial-state} = \\
\quad \lambda q_1.\lambda q_2.\cdots\lambda q_n.\lambda z.Kz \\
\quad\quad \langle Eq(q_1, true),\ Eq(q_2, \mathit{false}),\ Eq(q_3, \mathit{false}),\ \ldots,\ Eq(q_n, \mathit{false})\rangle \\
\textit{copy-state} = copy_n
\end{array}$$

Note that in a type faithfully encoding a machine state, only *one* of the q_i has the type of *true*, and the rest have the type of *false*. We now define a predicate giving the type output of *true* when applied to a state coding acceptance:

$$
\begin{array}{l}
accept? = \\
\quad \lambda state.\lambda x.\lambda y.\lambda z.Kz \\
\quad\quad \lambda q_1.\lambda q_2.\cdots\lambda q_n.\lambda acc. \\
\quad\quad\quad \langle state\ q_1\, q_2\ \cdots q_n, \\
\quad\quad\quad Eq(acc, \mathrm{or}\ q_{\ell+1}(\mathrm{or}\ q_{\ell+2}(\mathrm{or}\cdots(\mathrm{or}\ q_{m-1}\, q_m)\cdots))), \\
\quad\quad\quad acc\ x\ y\rangle
\end{array}
$$

The type of the "answer," i.e., the functional application *accept? state*, is the type of the ML term $\lambda x.\lambda y.\lambda z.Kz$, subject to the constraints that follow. The expression $state\ q_1\, q_2\ \cdots q_n$ forces the types of the q_i to unify with Boolean values encoded in the type of *state*. The type of *acc* is then constrained to be that of *true* or *false*, depending on the type of the Boolean expression. The final constraint $acc\ x\ y$ forces x and y in the "answer" to unify if the Boolean formula typed as *true*.

A predicate *reject?* is defined similarly.

5.6 Generating an exponential amount of blank tape.

In coding up an initial ID of the Turing Machine in ML, we need to generate the exponential space in which the exponential time machine can run. We represent tape as a list, and begin by defining *nil*:

$$nil \quad = \quad \lambda z.z$$

Now we use function composition to generate an exponential amount of tape using a polynomial-sized expression. Let $p(n)$ be a polynomial in the length of the input to the Turing Machine; in the following, we explicitly include the `let` syntax to emphasize the power of polymorphism needed.

$$
\begin{array}{l}
exponential\text{-}tape = \\
\quad \texttt{let}\ zero_0 = \lambda tape.[zero;\ tape]\ \texttt{in} \\
\quad\quad \texttt{let}\ zero_1 = \lambda tape.zero_0(zero_0\, tape)\ \texttt{in} \\
\quad\quad\quad \texttt{let}\ zero_2 = \lambda tape.zero_1(zero_1\, tape)\ \texttt{in} \\
\quad\quad\quad\quad \cdots \\
\quad\quad\quad\quad \texttt{let}\ zero_{p(n)} = \lambda tape.zero_{p(n)-1}(zero_{p(n)-1}\, tape)\ \texttt{in} \\
\quad\quad\quad\quad\quad zero_{p(n)}\ nil
\end{array}
$$

The nested `let`-expression then `let`-reduces to the ML term

$$[zero;\ [zero;\ [zero;\ [zero;\cdots[zero;\ nil]\cdots]]]]$$

where we have $2^{p(n)}$ *zero*es. By the use of pairing, we can also define an expression *exponential-tape-with-input* to be the list *exponential-tape* appended to the list encoding the input to the Turing Machine.

5.7 Turing Machine IDs.

We represent a Turing Machine ID by a type

$$\underline{state} \longrightarrow \underline{left} \longrightarrow \underline{right} \longrightarrow a \longrightarrow a,$$

where $\underline{state}$, $\underline{left}$, and $\underline{right}$ are type metavariables representing more complicated type structures encoding, respectively, the state of the machine (as described in Section 5.5), and lists constructed with *pair* representing the contents of the tape to the left and right of the tape head of the machine. We imagine that the tape head is currently reading the first cell on the list *right*.

$$\begin{aligned}
&\textit{initial-ID} = \\
&\quad \lambda state.\lambda left.\lambda right.\lambda z.Kz \\
&\qquad \langle Eq(state, \textit{initial-state}), \\
&\qquad\ Eq(left, \textit{exponential-tape}), \\
&\qquad\ Eq(right, \textit{exponential-tape-with-input})\rangle
\end{aligned}$$

5.8 State transition function.

Computing the next state of the Turing Machine is simply a Boolean function

$$\sigma(q_1, q_2, \ldots, q_n, z, o) = (t_1, t_2, \ldots, t_n),$$

where exactly one of the q_i is *true*, indicating that the machine is in state q_i, and either z or o is true, indicating what value is being read. A circuit to compute σ would form all the conjuncts $q_i \wedge z$, $q_i \wedge o$, partition the Boolean outputs of these $2n$ *and* gates into disjoint sets S_i, $1 \leq i \leq n$, and disjoin each S_i to generate the value of t_i. Viewed as a circuit, each input q_i has outdegree 2, the outdegree of z and o is n, and the outdegree of each conjunct is 1. Our simulation of σ thus uses the *fanout* gate to generate that many copies of each variable to realize the circuit faithfully.

$$\begin{aligned}
&\textit{next-state} = \\
&\quad \lambda state.\lambda cell. \\
&\qquad \lambda t_1.\lambda t_2.\cdots\lambda t_n.\lambda w.Kw \\
&\qquad\quad \lambda state_1.\lambda state_2. \\
&\qquad\quad \lambda cell_1.\lambda cell_2.\cdots\lambda cell_n. \\
&\qquad\quad \lambda z_1.\lambda z_2.\cdots\lambda z_n.\lambda o_1.\lambda o_2.\cdots\lambda o_n. \\
&\qquad\quad \lambda q_1^{(1)}.\lambda q_2^{(1)}.\cdots\lambda q_n^{(1)}.\lambda q_1^{(2)}.\lambda q_2^{(2)}.\cdots\lambda q_n^{(2)}. \\
&\qquad\qquad \langle \textit{copy-state}\ state\ state_1\ state_2, \\
&\qquad\qquad \textit{copy}_{2,n}\ cell\ cell_1\ cell_2\ \cdots\ cell_n, \\
&\qquad\qquad state_1\ q_1^{(1)}\ q_2^{(1)}\ \cdots\ q_n^{(1)},\ state_2\ q_1^{(2)}\ q_2^{(2)}\ \cdots\ q_n^{(2)}, \\
&\qquad\qquad cell_1\ z_1\, o_1,\ cell_2\ z_2\, o_2,\ \ldots,\ cell_n\ z_n\, o_n, \\
&\qquad\qquad Eq(t_1, \phi_1),\ Eq(t_2, \phi_2), \ldots,\ Eq(t_n, \phi_n)\rangle
\end{aligned}$$

The formula ϕ_i computes whether state q_i is reached at the next transition: it is just a Boolean expression using *or* and *and* gates, where we write the conjunction of the Boolean variables q_i and z (respectively, o) as *and* $q_i^{(1)} z_i$ (respectively, *and* $q_i^{(2)} o_i$). Note that just the right number of copies of each input have been provided via state and cell copying, and that *state* and *cell* are only used for replication, and not Boolean calculation.

5.9 Computing the new value of the tape cell being read.

The construction of the ML expression giving the new value written on the currently-read tape cell is virtually identical to the expression for giving the next state, detailed above. The only difference is that we have fewer Boolean outputs.

$$
\begin{aligned}
&\textit{new-cell} = \\
&\quad \lambda state.\lambda cell. \\
&\qquad \lambda f.\lambda g.\lambda h.Kh \\
&\qquad\quad \lambda state_1.\lambda state_2. \\
&\qquad\quad \lambda cell_1.\lambda cell_2.\cdots\lambda cell_n. \\
&\qquad\quad \lambda z_1.\lambda z_2.\cdots\lambda z_n.\lambda o_1.\lambda o_2.\cdots\lambda o_n. \\
&\qquad\quad \lambda q_1^{(1)}.\lambda q_2^{(1)}.\cdots\lambda q_n^{(1)}.\lambda q_1^{(2)}.\lambda q_2^{(2)}.\cdots\lambda q_n^{(2)}. \\
&\qquad\qquad \langle \textit{copy-state}\ state\ state_1\ state_2, \\
&\qquad\qquad copy_{2,n}\ cell\ cell_1\, cell_2\ \cdots\ cell_n, \\
&\qquad\qquad state_1\ q_1^{(1)}\, q_2^{(1)}\ \cdots\ q_n^{(1)},\ state_2\ q_1^{(2)}\, q_2^{(2)}\ \cdots\ q_n^{(2)}, \\
&\qquad\qquad cell_1\ z_1\, o_1,\ cell_2\ z_2\, o_2,\ \ldots,\ cell_n\ z_n\, o_n, \\
&\qquad\qquad Eq(f, \phi_{\mathrm{zero?}}), Eq(g, \phi_{\mathrm{one?}})\rangle
\end{aligned}
$$

The expressions $\phi_{\mathrm{zero?}}$ and $\phi_{\mathrm{one?}}$ are Boolean formulas indicating whether a zero or a one is written in the tape cell. Again, care must be taken to use each input "copy" once.

5.10 Turing machine transition function.

Now that we have encoded the computation of the next state of the Turing Machine, and the value to be written in the currently-read tape cell, it remains only to code left and right moves of the tape head. Observe that it is easy to encode a function *move* which computes a pair [*left?; right?*], where *left?* has the type of *true* if the Turing Machine is to move the head left, and *right?* has the type of *true* if the Turing Machine is to move the head right. The code for this computation is virtually identical to the definition of *new-cell* (Section 5.9), only with different Boolean functions.

More problematic, however, is *what* to do with the Boolean pair coding the head movement. The computation of the state and written value is in some sense "uniform," but the *type* encoding the next ID of the Turing Machine depends on the type returned by *move*. It is straightforward enough to compute the next ID if we knew in advance if the tape head was to move left, and another ID if the tape head was to move right. The challenge is to choose between them. To solve this problem, we have to introduce a type simulation of conditional assignment:

$$cond \quad = \quad \lambda p.\lambda \overline{p}.\lambda u.\lambda x.\lambda y.\lambda z.K\ z\ \langle p\ u\ x,\ \overline{p}\ u\ y\rangle$$

Note that typing *cond true false u I Eq* will unify the type of u with the type of I, since they are so constrained by the typing of *true u I*, while *false u Eq* introduces no such constraints, since *false* is a function which does not care about the respective types of its two curried arguments. Similarly, we note that typing *cond false true u I Eq* will unify u and Eq. The unification logic employed in this construction is exactly like a multiplexer.

The importance of typing an expression like *cond true false u I Eq* is clearly the constraint introduced on the type of u; when u is a lambda-bound variable, this constraint serves as a conditional type assignment. It is a curious situation that when proving a theorem about the typing mechanism of a purely functional programming language, we are led to a simulation of assignment to variables.

We now code a transition of the Turing Machine, if the head is moving right:

$$\begin{aligned}
&\textit{delta-right} = \\
&\quad \lambda \textit{old-ID}. \\
&\qquad \lambda \textit{new-state}.\lambda \textit{new-left}.\lambda \textit{new-right}.\lambda z.K\ z \\
&\qquad\quad \lambda \textit{state}.\lambda \textit{left}.\lambda \textit{right}.\lambda \textit{cell}. \\
&\qquad\quad \lambda \textit{state}_1.\lambda \textit{state}_2.\lambda \textit{cell}_1.\lambda \textit{cell}_2. \\
&\qquad\qquad \langle \textit{old-ID state left right}, \\
&\qquad\qquad \textit{right cell new-right}, \\
&\qquad\qquad \textit{copy-state state}\ \textit{state}_1\ \textit{state}_2, \\
&\qquad\qquad \textit{copy-cell cell}\ \textit{cell}_1\ \textit{cell}_2, \\
&\qquad\qquad Eq(\textit{new-state}, \textit{next-state}\ \textit{state}_1\ \textit{cell}_1), \\
&\qquad\qquad Eq(\textit{new-left}, [\textit{new-cell}\ \textit{state}_2\ \textit{cell}_2; \textit{left}])\rangle
\end{aligned}$$

Notice that the term *right cell new-right* simulates the breaking of the right hand side of the tape into the cell being read (*cell*) and the rest of the tape to the right (*new-right*).

The function *delta-left* can be defined similarly. Next, we define a function which will make copies of Turing Machine IDs. Observe that only the state and currently read tape cell are actually copied, so that the copies share the remaining tape contents.

$$\begin{aligned}
&\textit{copy-ID} = \\
&\quad \lambda \textit{ID}. \\
&\qquad \lambda \textit{ID}_1.\lambda \textit{ID}_2.\lambda z.K z \\
&\qquad\quad \lambda \textit{state}.\lambda \textit{left}.\lambda \textit{right}.\lambda \textit{cell}.\lambda \textit{tape}. \\
&\qquad\quad \lambda \textit{state}_1.\lambda \textit{state}_2.\lambda \textit{cell}_1.\lambda \textit{cell}_2. \\
&\qquad\qquad \langle \textit{ID state left right}, \\
&\qquad\qquad \textit{right cell tape}, \\
&\qquad\qquad \textit{copy-state state}\ \textit{state}_1\ \textit{state}_2, \\
&\qquad\qquad \textit{copy-cell cell}\ \textit{cell}_1\ \textit{cell}_2,
\end{aligned}$$

$$
\begin{aligned}
&Eq(ID_1,\ \lambda s.\lambda \ell.\lambda r.\lambda z.z),\\
&Eq(ID_2,\ \lambda s.\lambda \ell.\lambda r.\lambda z.z),\\
&ID_1\ state_1\ left\ [cell_1;\ tape],\\
&ID_2\ state_2\ left\ [cell_2;\ tape]\rangle
\end{aligned}
$$

We now use the conditional assignment *cond* to choose either *delta-left* or *delta-right*, and code the transition function as:

$$
\begin{aligned}
&delta =\\
&\quad \lambda old\text{-}ID.\\
&\qquad \lambda new\text{-}state.\lambda new\text{-}left.\lambda new\text{-}right.\lambda z.K\ z\\
&\qquad\quad \lambda state.\lambda left.\lambda right.\lambda cell.\\
&\qquad\quad \lambda ID_1.\lambda ID_2.\lambda\Delta.\lambda left?.\lambda right?.\\
&\qquad\qquad \langle copy\text{-}ID\ old\text{-}ID\ ID_1\ ID_2,\\
&\qquad\qquad ID_1\ state\ left\ right,\\
&\qquad\qquad right\ cell,\\
&\qquad\qquad (move\ state\ cell)\ left?\ right?,\\
&\qquad\qquad cond\ left?\ right?\ \Delta\ delta\text{-}left\ delta\text{-}right,\\
&\qquad\qquad (\Delta\ ID_2)\ new\text{-}state\ new\text{-}left\ new\text{-}right\rangle
\end{aligned}
$$

5.11 The simulation: Finale.

The innermost sequence of `let` expressions brings the simulation to its conclusion:

$$
\begin{aligned}
&\texttt{let}\ \delta_0 = delta\ \texttt{in}\\
&\quad \texttt{let}\ \delta_1 = \lambda ID.\delta_0(\delta_0\ ID)\ \texttt{in}\\
&\qquad \texttt{let}\ \delta_2 = \lambda ID.\delta_1(\delta_1\ ID)\ \texttt{in}\\
&\qquad\quad \texttt{let}\ \delta_3 = \lambda ID.\delta_2(\delta_2\ ID)\ \texttt{in}\\
&\qquad\qquad \cdots\\
&\qquad\qquad \texttt{let}\ \delta_{p(n)} = \lambda ID.\delta_{p(n)-1}(\delta_{p(n)-1}\ ID)\ \texttt{in}\\
&\qquad\qquad\quad K\ I\ (\lambda state.\lambda z.\ K\ z\\
&\qquad\qquad\qquad\qquad \langle(\delta_{p(n)}\ initial\text{-}ID)\ state,\\
&\qquad\qquad\qquad\qquad (reject?\ state)\ Eq\ I\rangle)
\end{aligned}
$$

Remember that (*reject? state*) returns *true*: $a \longrightarrow a \longrightarrow b \longrightarrow b$ or *false*: $a \longrightarrow b \longrightarrow c \longrightarrow c$; in the case of the former, $Eq\colon a \longrightarrow a \longrightarrow a$ and $I\colon a \longrightarrow a$ will be forced to be unified, causing a mistyping. If no mistyping occurs, then the type of the entire expression is that of the identity function.

Theorem 5.23 *For every integer $k \geq 1$, the problem of deciding whether a core ML expression is typable is logspace complete in the class of languages* DTIME(2^{n^k}).

Proof. Recall that DTIME$(f(n))$ is the class of languages L with deterministic Turing Machine recognizer M_L such that M_L accepts or rejects input x in no more than $f(|x|)$

steps. By Corollary 3.16, we know that the typable core ML expressions form a language contained in DTIME(2^n) $\subseteq$ DTIME(2^{n^k}). Now suppose for some language $L \subseteq \{0,1\}^*$ that $L \in$ DTIME(2^{n^k}). Let M be a Turing Machine recognizing L, so that for any input $x \in \{0,1\}^*$, M accepts or rejects x in no more than $2^{|x|^k}$ steps. Given $p(n) = n^k$, the above construction details a polynomial time reduction such that M accepts x iff the ML formula $\Psi_{M,x}$ is typable, where $|\Psi_{M,x}| \leq c|x|^k$ for some constant $c > 0$ depending on M.

The reduction is not only polynomial time, but may also be carried out in $\log n$ work space by a Turing Machine transducer (for details of these definitions, see [HU]). The output $\Psi_{M,x}$ requires n^k nested `let` definitions to define the exponential tape, and n^k nested `let` definitions to define the transition function, where the lengths of the `let` bindings are constant. Observe that the number of variable identifiers in $\Psi_{M,x}$ may be kept constant, since the identifiers δ_i and $zero_i$, $0 \leq i \leq n^k$ need not be unique. The work tape of the transducer is used merely to count the *number* of nested definitions of δ_i and $zero_i$ that need be output; all other details of the output can be managed by the finite state control. This counting can be realized in $O(\log n)$ bits, to be represented in $\log n$ space given a suitably large work tape alphabet. ∎

Our generic simulation leads to completeness lower bounds, but also to unconditional lower bounds in the style of [Mey72, SM73]. In particular, consider the proof of the previous theorem with $k = 1$. In this case $|\Psi_{M,x}| \leq c|x|$ for some constant $c > 0$ depending on M. Using an argument analogous to that of Theorem 13.15 from [HU] we have:

Theorem 5.24 *Any algorithm recognizing the typable core ML expressions of length n is at least of time complexity bc^n infinitely often, for some constants $b > 0$ and $c > 1$.*

5.12 Some comments on the lower bound.

We have provided a simple proof that deciding ML typability is of intrinsic exponential difficulty. The proof was just a computer program written in ML, whose principal type simulated an exponential number of moves by an arbitrary Turing Machine; by changing the program slightly, we could force a mistyping precisely when the Turing Machine rejected its input.

The only parts of the above construction where ML `let`-polymorphism is absolutely necessary are where we use exponential function composition: in constructing the exponential tape of zeroes, and in the construction of the transition function, detailed in Sections 5.6 and 5.11. The other uses of `let` are mere notational conveniences, serving only to make the construction more readable; we could remove them by `let`-reduction (i.e., reinstantiating several copies of the code) without the resulting ML formula blowing up exponentially, so that we no longer have a polynomial reduction.

It is technically possible to remove the use of exponential function composition for coding up the tape as in Section 5.6, so that the *only* use of exponential function composition is to compose the transition function. We code up an exponential amount of tape in Section 5.6 because in an exponential number of state transitions, the tape head can only move an exponential number of steps to the left or right. The tape is "manufactured" *in toto* before it is used, and it is clear in the simulation that the tape head never "falls off" the end of a tape—in other words, enough tape has been coded. However, it is possible to modify the definition of the tape symbols to be either zero, one, or a special *end-of-tape* symbol. In addition, we could modify the encoding of the transition function *delta* so that, if it detects an end-of-tape symbol, it codes up another cell of tape, this checking implemented using conditional assignment.

Because of the generic reduction detailed here, lower bounds on typability of extensions to the ML type discipline—or extensions to the expressive power of the typed lambda calculus—can probably be established merely by considering how succinctly functions can be composed. The following proposition provides an example:

Proposition 5.25 *Given a strongly normalizing lambda term M, the problem of determining whether the normal form of M is first-order (simply) typable is* DTIME($f(n)$)*-hard, for* any *total recursive function $f(n)$.*

Proof. Take any Turing Machine M which halts in $f(n)$ steps on an input x of length n, and consider the typing of the lambda term $\overline{f}\ \overline{n}\ \Delta\ x$, where $\overline{f}$ is the Church-numeral encoding of f, $\overline{n}$ is the Church numeral for n, Δ encodes the transition function for M, and x encodes the input. (The function composition we speak of above takes place in the reduction of $\overline{f}\ \overline{n}$.) We force a mistyping if the typing of this term encodes rejection of the input. ∎

6 Conclusions.

The Damas-Milner 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. Since the problem of recognizing typable ML expressions requires exponential time, no substantial efficiency can be expected in the worst case.

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 3.15, 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. Both of these explanations deserve further investigation.

In addition to the syntax of core ML, most ML dialects have a fixpoint combinator whose presence does not affect the complexity of type reconstruction. However, one extension of ML is based on polymorphic typing of the fixpoint combinator [Myc84]. Type reconstruction for this extension has been characterized in [Hen89, KTU89] as equivalent to semi-unification, and has recently been shown to be undecidable [KTU90b]. The analysis of semi-unification, under an acyclicity condition, is the basis for the proof of the exponential lower bounds in [KTU90a].

One promising direction for further work is to apply the unification techniques here as well as the theory of semi-unification to other typing problems, e.g., see [Lei83, Mit88, GR88, Hen89, KTU89]. The major remaining open problem is the decidability question (or even any nontrivial lower bound) for type reconstruction in System F or second order typed lambda calculus [Gir72, Rey74].

Acknowledgements: Earlier versions of this work appeared in [KM89, Mai90]. 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 Gerd Hillebrand and Ken Perry for many helpful discussions, and acknowledge the support of the IBM T.J. Watson Research Center.

References

[AS85] H. Abelson and G. J. Sussman. **Structure and Interpretation of Computer Programs.** MIT Press, 1985.

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

[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 International Conference 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.

[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.

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

[GR88] P. Giannini and S. Ronchi Della Rocca. *Characterization of typings in polymorphic type discipline.* In **Proceedings of the 3-rd IEEE Symposium on Logic in Computer Science**, pages 61–70, July 1988.

[Gir72] J.-Y. Girard, *Interprétation Fonctionelle et Elimination des Coupures de l'Arithmetique d'Ordre Superieur.* Thèse de Doctorat d'Etat, Université de Paris VII, 1972.

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

[Hen89] F. Henglein. *Polymorphic type inference and semi-unification.* New York University Technical Report 443, May 1989. Also see **Proceedings of the ACM Symposium on Lisp and Functional Programming Languages**, pages 184–197, July 1988.

[Hin69] R. Hindley. *The principal type scheme of an object in combinatory logic.* **Transactions of the American Mathematical Society** 146:29–60,1969.

[HU] J. E. Hopcroft and J. D. Ullman. **Introduction to Automata Theory, Languages, and Computation.** Addison Wesley 1979.

[HW88] P. Hudak and P. L. Wadler, editors. *Report on the Functional Programming Language Haskell.* Yale University Technical Report YALEU/DCS/RR656, 1988.

[KM89] P. C. Kanellakis and J. C. Mitchell. *Polymorphic unification and ML typing.* Brown University Technical Report CS-89-40, August 1989. Also in **Proceedings of the 16-th ACM Symposium on the Principles of Programming Languages**, pages 105–115, January 1989.

[KTU89] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. *Computational consequences and partial solutions of a generalized unification problem.* In **Proceedings of the 4-th IEEE Symposium on Logic in Computer Science**, pages 98–105, July 1989.

[KTU90a] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. *ML Typability is Dexptime-Complete.* To appear in the **Proceedings of the 15-th Colloquium on Trees in Algebra and Programming**, May 1990. (See also Boston University Technical Report, October 1989).

[KTU90b] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. *Undecidability of the semi-unification problem.* To appear in the **Proceedings of the 22nd ACM Symposium on Theory of Computing**, May 1990. (See also Boston University Technical Report, October 1989).

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

[Mai90] H. G. Mairson. *Deciding ML typability is complete for deterministic exponential time.* In **Proceedings of the 17-th ACM Symposium on the Principles of Programming Languages**, pages 382–401, January 1990.

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

[Mey72] A. R. Meyer. *Weak S1S cannot be decided.* In **Notices Amer. Math. Soc.**, 19, A-598, Abstract 72T-E67, 1972. (See also, Lecture Notes in Mathematics 453, Springer Verlag, pages 132–154, 1975.)

[Mil78] R. Milner. *A theory of type polymorphism in programming.* **Journal of Computer and System Sciences** 17, pages 348-375, 1978.

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

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

[Mit90] J. C. Mitchell. *Type systems for programming languages.* To appear as a chapter in the **Handbook of Theoretical Computer Science**, van Leeuwen et al., eds. North-Holland, 1990.

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

[Myc84] A. Mycroft. *Polymorphic types schemes and recursive definitions.* In **Proceedings International Symposium on Programming**, M. Paul and B. Robinet eds, Lecture Notes in Computer Science 167, Springer Verlag, pages 217–228, 1984.

[PW78] M. S. Paterson and M. N. Wegman. *Linear unification.* **Journal of Computer and System Sciences** 16, pages 158–167, 1978.

[Rey74] J. C. Reynolds. *Towards a theory of type structure.* In **Proceedings of the Paris Colloquium on Programming**, Lecture Notes in Computer Science 19, Springer Verlag, pages 408–425, 1974.

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

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

[Tur85] D. A. Turner. *Miranda: A non-strict functional language with polymorphic types.* In **IFIP International Conference on Functional Programming and Computer Architecture**, Nancy, Lecture Notes in Computer Science 201, pages 1–16, Springer-Verlag, 1985.

[Wan87] M. Wand. *A simple algorithm and proof for type inference.* **Fundamenta Informaticae** 10 (1987).

A Type inference rules and algorithm

We first summarize our inference system for core ML. Next, since our inference system is not the traditional presentation of ML typing, we prove that our typing rules are equivalent to the "standard" Damas-Milner typing rules [DM82, Mil78]. For more details we refer to [Mit90]; see also [MH88]. Finally, we describe the Damas-Milner algorithm, which is commonly used in ML implementations.

A.1 A definition of ML typing

We will write $\vdash_{KMM} \Gamma \triangleright M\colon \tau$ for typing assertions about core ML expressions provable using the following axiom and inference rules.

$$(var) \qquad \Gamma \oplus x\colon \tau_1 \triangleright x\colon \tau_1$$

$$(abs) \qquad \frac{\Gamma \oplus x\colon \tau_1 \triangleright M\colon \tau_2}{\Gamma \triangleright (\lambda x.M)\colon \tau_1 \rightarrow \tau_2}$$

$$(app) \qquad \frac{\Gamma \triangleright M\colon \tau_1 \rightarrow \tau_2, \quad \Gamma \triangleright N\colon \tau_1}{\Gamma \triangleright MN\colon \tau_2}$$

where $\Gamma \oplus x\colon \tau_1$ is the result of removing any statement about x from Γ and adding $x\colon \tau_1$.

The rules up to this point are often called the *Curry typing rules,* after H. B. Curry [CF58]. The following inference rule lets us type `let` expressions.

$$(let) \qquad \frac{\Gamma \triangleright M\colon \tau_1, \ \Gamma \triangleright [M/x]N\colon \tau}{\Gamma \triangleright \texttt{let}\ x = M\ \texttt{in}\ N\colon \tau}$$

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 typing assumption about M is only needed in the event that x does not appear free in N.

Substitution on typing assertions and the notion of instance are defined in Section 2.2. In reading the subsequent equivalence proof, it is useful to keep in mind that if $\Gamma \triangleright M\colon \tau$ is provable, and $\Gamma' \triangleright M\colon \tau'$ is an instance of this assertion, then $\Gamma' \triangleright M\colon \tau'$ is also a provable typing assertion (by Proposition 2.3).

As defined in Section 2.2, a typing assertion about M is principal if it is provable and has all other provable assertions about M as instances (see also Proposition 2.4). In proving equivalence with the Damas-Milner typing rules for ML, it will be useful to have an additional form of principal typing. We will say that a provable typing assertion $\Gamma \triangleright M\colon \tau$ is Γ-*principal* if for every other provable typing assertion $\Gamma \triangleright M\colon \tau'$ there is a substitution S affecting only type variables not in Γ such that $\tau' = S\tau$. Using well-known properties of first-order substitution and unification (as described in [PW78], for example), it is easy to show that principal typings give us Γ-principal typings.

Lemma A.26 *Suppose $\Gamma \triangleright M\colon \tau$ is a principal typing for M and $\Gamma' \triangleright M\colon \tau'$ is provable. Then $\Gamma' = S\Gamma \cup \Gamma''$ for some substitution S which only affects type variables in Γ, and $\Gamma' \triangleright M\colon S\tau$ is a Γ'-principal typing for M.*

A.2 Equivalence with Damas-Milner typing rules

The Damas-Milner typing rules for ML, given in [DM82], use an extended notion of type expression called a *type scheme.* More precisely, the *types* are type expressions of the form

$$\tau ::= \ t \,|\, \tau \longrightarrow \tau'$$

as defined in Section 2.2, and *type schemes* are expressions of the form

$$\sigma ::= \ \forall t_1 \ldots \forall t_k.\, \tau,$$

where τ is a type. We consider types as a special case of type schemes, arising from the special case of $k = 0$ quantifiers. Since $\forall$ is a binding operator, we consider $\forall t.\sigma = \forall s.[s/t]\sigma$, provided s not free in σ.

In this subsection we use the metavariable τ for types and σ for type schemes.

We will write $\vdash_{DM} \Gamma \triangleright M\colon \sigma$ if this typing assertion is provable using the Damas-Milner rules right below. In these rules, a type assignment Γ may contain typing assumptions $x\colon \sigma$ giving types or type schemes to term variables.

$(var)_{DM}$ $\qquad \Gamma \oplus x\colon \sigma \triangleright x\colon \sigma$

$(inst)_{DM}$ $\qquad \dfrac{\Gamma \triangleright M\colon \forall t_1 \ldots \forall t_k.\, \sigma}{\Gamma \triangleright M\colon [\tau_1, \ldots, \tau_k/t_1, \ldots, t_k]\sigma}$

$(gen)_{DM}$ $\qquad \dfrac{\Gamma \triangleright M\colon \sigma}{\Gamma \triangleright M\colon \forall t.\, \sigma}$, t not free in Γ

$(abs)_{DM}$ $\qquad \dfrac{\Gamma \oplus x\colon \tau_1 \triangleright M\colon \tau_2}{\Gamma \triangleright (\lambda x.M)\colon \tau_1 \rightarrow \tau_2}$

$(app)_{DM}$ $\qquad \dfrac{\Gamma \triangleright M\colon \tau_1 \rightarrow \tau_2, \quad \Gamma \triangleright N\colon \tau_1}{\Gamma \triangleright MN\colon \tau_2}$

$(let)_{DM}$ $\qquad \dfrac{\Gamma \triangleright M\colon \sigma,\ \Gamma \oplus x\colon \sigma \triangleright N\colon \tau}{\Gamma \triangleright \texttt{let}\ x = M\ \texttt{in}\ N\colon \tau}$

The only slight variance with the rules in [DM82] is a weakening of $(inst)_{DM}$, but this is clearly unimportant in the presence of $(gen)_{DM}$ (*cf.* [MH88]).

The remainder of this section will be devoted to showing that the two systems are equivalent for proving typing assertions of the form $\Gamma \triangleright M\colon \tau$, where Γ contains no type schemes and τ is simply a type (*i.e.*, $\forall$ does not appear in τ).

To see that this correspondence is sufficient, observe that a "normal ML program" is an expression with no free variables. For closed M, a typing assertion $\Gamma \triangleright M\colon \sigma$ is provable (in either system) iff we can prove the assertion $\emptyset \triangleright M\colon \sigma$ with empty type assignment. (This property holds for almost any type system, and lemmas to this effect may be found in [Mit88, Mit90], for example.) Thus when we consider closed ML expressions, type assignments Γ containing type schemes are irrelevant. Moreover, it is easy to see that in the Damas-Milner system, $\vdash_{DM} \Gamma \triangleright M\colon \forall t_1 \ldots \forall t_k.\, \tau$ iff $\vdash_{DM} \Gamma \triangleright M\colon \tau$, provided we choose bound variables $t_1, \ldots, t_k$ not occurring in Γ. Therefore, in studying typability, it suffices to consider types instead of type schemes.

We begin by noting that for the Damas-Milner system, it is easy to verify that if $\vdash_{DM}$ $\Gamma \triangleright M\colon \sigma$, then any instance of this typing assertion is also provable. This is Proposition 2 of [DM82]. We now show two technical properties of the Damas-Milner typing system. The first lemma gives a "normal form" property for typing derivations.

Lemma A.27 *Let Γ be a type assignment and let τ be any type (i.e., without $\forall$). Suppose that either Γ contains no type schemes or M is not a variable. If $\vdash_{DM} \Gamma \triangleright M\colon \forall t_1 \ldots \forall t_k.\, \tau$, then $\vdash_{DM} \Gamma \triangleright M\colon \tau$ by a proof whose last step is either the axiom $(var)_{DM}$ or an application of one of the proof rules $(abs)_{DM}$, $(app)_{DM}$, or $(let)_{DM}$.*

Proof. The proof is by cases (not induction), depending on the structure of terms. For a variable x, every proof must begin with the axiom $\Gamma \oplus x\colon \tau \triangleright x\colon \tau$. But this is all we can prove, since every type in the type assignment is assumed $\forall$-free: we cannot apply $(gen)_{DM}$ nontrivially since every type variable free in τ obviously appears in the type assignment $\Gamma \oplus x\colon \tau$. The remaining cases all resemble each other. We will show only the application case and leave the remaining details to the reader.

Consider a provable typing assertion $\Gamma \triangleright MN\colon \tau$. The proof of this assertion must involve some application of the rule

$$(app)_{DM} \qquad \frac{\Gamma \triangleright M\colon \tau_1 \rightarrow \tau_2, \quad \Gamma \triangleright N\colon \tau_1}{\Gamma \triangleright MN\colon \tau_2}$$

possibly followed by rules $(gen)_{DM}$ and $(inst)_{DM}$. Since these rules may only lead to the substitution of type expressions for type variables in τ_2, we have $\tau = S\tau_2$ for some substitution S. (This is easily verified, but the reader wishing further explanation may consult [Mit88, Lemma 12].) Since the provable typings are closed under substitution, both $\Gamma \triangleright M\colon S(\tau_1 \rightarrow \tau_2)$ and $\Gamma \triangleright N\colon S\tau_1$ must be provable, giving us a proof of $\Gamma \triangleright MN\colon \tau$ by rule $(app)_{DM}$. ∎

Lemma A.28 *Let Γ be any type assignment, possibly containing type schemes. Assume $\vdash_{DM} \Gamma \triangleright M\colon \forall t_1 \ldots \forall t_k.\, \tau$ and that for every provable typing assertion $\Gamma \triangleright M\colon \tau'$, the type τ' has the form*

$$\tau' = [\tau_1, \ldots, \tau_k/t_1, \ldots, t_k]\tau.$$

Then $\vdash_{DM} \Gamma \triangleright [M/x]N\colon \tau''$ iff $\vdash_{DM} \Gamma \oplus x\colon (\forall t_1 \ldots \forall t_k.\, \tau) \triangleright N\colon \tau''$.

Proof. The proof is by induction on the structure of N. The variable case involves reasoning about sequences of $(inst)_{DM}$ and $(gen)_{DM}$ proof steps, as in [Mit88, Lemma 12]. The main idea is that $\vdash_{DM} \Gamma \triangleright M\colon \tau''$ iff $\tau'' = [\tau_1, \ldots, \tau_k/t_1, \ldots, t_k]\tau$ iff the typing assertion $\Gamma \triangleright M\colon \tau''$ may be obtained from $\Gamma \triangleright M\colon \forall t_1 \ldots \forall t_k.\, \tau$ by a series of applications of $(inst)_{DM}$ and $(gen)_{DM}$. But then this same series of proof steps may be used to obtain $\Gamma \oplus x\colon (\forall t_1 \ldots \forall t_k.\, \tau) \triangleright x\colon \tau''$ from $\Gamma \oplus x\colon (\forall t_1 \ldots \forall t_k.\, \tau) \triangleright x\colon \forall t_1 \ldots \forall t_k.\, \tau$, and conversely.

The remaining cases are essentially alike. We will illustrate the argument using the `let` case, since this is slightly more involved than the others. By Lemma A.27, we need only consider proofs which end with rule $(let)_{DM}$. Therefore, we have $\vdash_{DM} \Gamma \triangleright [M/x](\texttt{let}\ y = N_1\ \texttt{in}\ N_2)\colon \tau''$ iff $\vdash_{DM} \Gamma \triangleright (\texttt{let}\ y = [M/x]N_1\ \texttt{in}\ [M/x]N_2)\colon \tau''$ iff by the last proof rule both $\vdash_{DM} \Gamma \triangleright [M/x]N_1\colon \sigma'$ and $\vdash_{DM} \Gamma \oplus y\colon \sigma' \triangleright [M/x]N_2\colon \tau''$. Since τ'' is not a type scheme, the inductive hypothesis implies that the second condition is equivalent to $\vdash_{DM} \Gamma \oplus x\colon \sigma \oplus y\colon \sigma' \triangleright N_2\colon \tau''$, where $\sigma = (\forall t_1 \ldots \forall t_k.\, \tau)$.

To apply the inductive hypothesis to $\Gamma \triangleright [M/x]N_1 \colon \sigma'$, we note that σ' must have the form $\sigma' = \forall s_1 \ldots \forall s_\ell . \tau'$. We assume without loss of generality that $s_1, \ldots, s_\ell$ do not occur free in Γ and remove the quantifiers by rule (*inst*). Using the inductive hypothesis on the typing assertion obtained in this way, we observe:

$$\begin{array}{lll} \vdash_{DM} \Gamma \triangleright [M/x]N_1 \colon \forall s_1 \ldots \forall s_\ell . \tau' & \text{iff} & \vdash_{DM} \Gamma \triangleright [M/x]N_1 \colon \tau' \\ & \text{iff} & \vdash_{DM} \Gamma \oplus x \colon \sigma \triangleright N_1 \colon \tau' \\ & \text{iff} & \vdash_{DM} \Gamma \oplus x \colon \sigma \triangleright N_1 \colon \forall s_1 \ldots \forall s_\ell . \tau' \end{array}$$

∎

Theorem A.29 *Let Γ be a type assignment with no type schemes, and let τ be any type. Then $\vdash_{DM} \Gamma \triangleright M \colon \tau$ iff $\vdash_{KMM} \Gamma \triangleright M \colon \tau$.*

Proof. We show the equivalence of the two systems by a double induction on the size of M, and the largest number of `let`-reduction steps needed to reduce M to `let`-normal form. Recall from Proposition 2.1 that every core ML expression may be reduced to `let`-normal form by contracting `let` redexes, and that there are no infinite sequences of such reductions. A double induction is necessary because our structural induction on `let`-expressions involves `let`-reduction, and then using an inductive hypothesis. Since reducing the term may increase its size, a simple induction on expression size will not work; however, the reduced term is a step closer to the complete development in `let`-normal form, and there cannot be an unbounded number of such steps. For terms without `let`, our induction becomes ordinary induction on the length of terms.

We consider each syntactic form of terms, and show that the two systems are equivalent for terms of this form. The cases for variable, application and abstraction are straightforward, using Lemma A.27.

Since the primary difference between the two systems is in the `let` rules, this is the main case of the proof. We prove each direction of the equivalence separately. If $\vdash_{KMM} \Gamma \triangleright \texttt{let}\ x = M\ \texttt{in}\ N \colon \tau$, then the derivation concludes with the rule

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

There are two cases to consider. If x does not occur free in N, then by the inductive hypothesis, we know that both hypotheses of this proof rule are Damas-Milner provable. It is easily seen that the Damas-Milner system allows for addition of irrelevant hypotheses; hence $\vdash_{DM} \Gamma \oplus x \colon \tau' \triangleright N \colon \tau$. Therefore, we have $\vdash_{DM} \Gamma \triangleright \texttt{let}\ x = M\ \texttt{in}\ N \colon \tau$ by rule $(let)_{DM}$.

The more difficult case is when x occurs free in N. Since we have shown that $\vdash_{KMM}$ has principal typings, in Proposition 2.4, we may assume by Lemma A.26 that $\Gamma \triangleright M \colon \tau'$ is the Γ-principal typing of M. This means that every KMM-provable typing assertion of the form $\Gamma \triangleright M \colon \tau''$ has $\tau'' = S\tau'$ for some substitution S which only affects type variables $t_1, \ldots, t_k$ that are free in τ' but not in Γ. This fact (Γ-principality) carries over to the Damas-Milner system by the induction hypothesis. By the inductive hypotheses, we also have $\vdash_{DM} \Gamma \triangleright M \colon \tau'$ and $\vdash_{DM} \Gamma \triangleright [M/x]N \colon \tau$. By rule $(gen)_{DM}$, we have $\vdash_{DM} \Gamma \triangleright M \colon \forall t_1 \ldots \forall t_k . \tau'$ and by Lemma A.28 $\vdash_{DM} \Gamma \oplus x \colon (\forall t_1 \ldots \forall t_k . \tau') \triangleright N \colon \tau$. This allows us to apply rule $(let)_{DM}$, concluding this direction of the proof.

For the converse, we assume $\vdash_{DM} \Gamma \triangleright \texttt{let}\ x = M\ \texttt{in}\ N\colon\tau$. By Lemma A.27, we may assume that the proof ends with the proof step

$$(let)_{DM} \qquad \frac{\Gamma \triangleright M\colon\sigma,\ \Gamma \oplus x\colon\sigma \triangleright N\colon\tau}{\Gamma \triangleright \texttt{let}\ x = M\ \texttt{in}\ N\colon\tau}$$

where σ has the form $\sigma = \forall t_1 \ldots \forall t_k.\, \tau'$. Without loss of generality, we may assume that $t_1, \ldots, t_k$ do not appear free in Γ. By $(inst)_{DM}$, we have $\vdash_{DM} \Gamma \triangleright M\colon\tau'$, and hence this assertion is KMM provable by the induction hypothesis. There is no loss of generality in assuming that $\Gamma \triangleright M\colon\tau'$ is a Γ-principal typing for M, since this does not affect the provability of $\Gamma \oplus x\colon\sigma \triangleright N\colon\tau$; we may always use $(inst)_{DM}$ on the variable x (see Lemma 1 and the discussion surrounding rule (subst) in [Mit88]). This gives us $\vdash_{DM} \Gamma \triangleright [M/x]N\colon\tau$ by Lemma A.28 and therefore $\vdash_{KMM} \Gamma \triangleright [M/x]N\colon\tau$ by the inductive hypothesis. We may now use rule (let) to finish the proof. ■

A.3 The Damas-Milner Algorithm

The algorithm *PT* given below in Figure 2 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. Since the algorithm may give incorrect results if a variable x is both `let`-bound and λ-bound, we assume that *the input to PT is a program with all bound variables renamed to be distinct.* Renaming bound variables in this way is commonly done in the lexical analysis phase of compilation, 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.

Proposition A.30 *If* $PT(M, \emptyset) = \Gamma \triangleright M\colon\tau$, *then* $\Gamma \triangleright M\colon\tau$ *is a provable typing statement.*

It follows, by Proposition 2.3, that every instance of $PT(M, \emptyset)$ is provable. Conversely, every provable typing for M is an instance of $PT(M, \emptyset)$.

Proposition A.31 *Suppose* $\Gamma \triangleright M\colon\tau$ *is a provable typing. Then* $PT(M, \emptyset)$ *succeeds and produces a typing with* $\Gamma \triangleright M\colon\tau$ *as an instance.*

$$
\begin{array}{ll}
PT(x, A) & = \textbf{if } A(x) = \Gamma \triangleright M\!: \sigma \textbf{ then } \Gamma \triangleright x\!: \sigma \\
& \quad \textbf{else } \{x\!: t\} \triangleright x\!: t \\[1em]
PT(MN, A) & = \\
& \textbf{let} \\
& \quad \Gamma_1 \triangleright M\!: \sigma = PT(M, A) \\
& \quad \Gamma_2 \triangleright N\!: \tau = PT(N, A), \\
& \qquad \text{with type variables renamed to be disjoint from those in } PT(M, A) \\
& \quad S = \mathit{UNIFY}(\{\, \alpha = \beta \mid x\!: \alpha \in \Gamma_1 \text{ and } x\!: \beta \in \Gamma_2 \,\} \cup \{\sigma = \tau \rightarrow t\}), \\
& \qquad \text{where } t \text{ is a fresh type variable} \\
& \textbf{in} \\
& \qquad S\Gamma_1 \cup S\Gamma_2 \triangleright MN\!: St \\[1em]
PT(\lambda x.M, A) & = \\
& \textbf{let } \Gamma \triangleright M\!: \tau = PT(M, A) \\
& \textbf{in} \\
& \quad \textbf{if } x\!: \sigma \in \Gamma \text{ for some } \sigma \\
& \qquad \textbf{then } \Gamma - \{x\!: \sigma\} \triangleright \lambda x.M\!: \sigma \rightarrow \tau \\
& \qquad \textbf{else } \Gamma \triangleright \lambda x.M\!: s \rightarrow \tau, \\
& \qquad\quad \text{where } s \text{ is a fresh type variable} \\[1em]
PT(\texttt{let } x = M \texttt{ in } N, A) & = \\
& \textbf{let } \Gamma_1 \triangleright M\!: \sigma = PT(M, A) \\
& \quad A' = A \cup \{x \mapsto \Gamma_1 \triangleright M\!: \sigma\} \\
& \quad \Gamma_2 \triangleright N\!: \tau = PT(N, A') \\
& \quad S = \mathit{UNIFY}(\{\, \alpha = \beta \mid y\!: \alpha \in \Gamma_1 \text{ and } y\!: \beta \in \Gamma_2 \,\}) \\
& \textbf{in } \; S\Gamma_1 \cup S\Gamma_2 \triangleright \texttt{let } x = M \texttt{ in } N\!: S\tau
\end{array}
$$

Figure 2: Algorithm PT to compute principal typing.