Deciding ML Typability is Complete for Deterministic Exponential Time

HARRY G. MAIRSON
Department of Computer Science
Brandeis University
Waltham, Massachusetts 02254

Abstract. A well known but incorrect piece of functional programming folklore is that ML expressions can be efficiently typed in polynomial time. In probing the truth of that folklore, various researchers, including Wand, Buneman, Kanellakis, and Mitchell, constructed simple counterexamples consisting of typable ML programs having length n, with principal types having $\Omega(2^{cn})$ distinct type variables and length $\Omega(2^{2^{cn}})$. When the types associated with these ML constructions were represented as directed acyclic graphs, their sizes grew as $\Omega(2^{cn})$. The folklore was even more strongly contradicted by the recent result of Kanellakis and Mitchell that simply deciding whether or not an ML expression is typable is PSPACE-hard.

We improve the latter result, showing that deciding ML typability is DEXPTIME-hard. As Kanellakis and Mitchell have shown containment in DEXPTIME, the problem is DEXPTIME-complete. The proof of DEXPTIME-hardness is carried out via a generic reduction: it consists of a very straightforward simulation of any deterministic one-tape Turing machine M with input x running in $O(c^{|x|})$ time by a polynomial-sized ML formula $\Phi_{M,x}$, such that M accepts x iff $\Phi_{M,x}$ is typable. The simulation of the transition function δ of the Turing Machine is realized uniquely through terms in the lambda calculus *without* the use of the polymorphic *let* construct. We use *let* for two purposes only: to generate an exponential amount of blank tape for the Turing Machine simulation to begin, and to compose an exponential number of applications of the ML formula simulating state transition.

It is purely the expressive power of ML polymorphism to succinctly express function composition which results in a proof of DEXPTIME-hardness. We conjecture that lower bounds on deciding typability for extensions to the typed lambda calculus can be regarded precisely in terms of this expressive capacity for succinct function composition.

To further understand this lower bound, we relate it to the problem of proving equality of type variables in a system of type equations generated from an ML expression with let-polymorphism. We show that given an oracle for solving this problem, deciding typability would be in PSPACE, as would be the actual computation of the principal type of the expression, were it indeed typable.

1 Introduction.

ML [Mi78][HMM86] is a well known functional programming language incorporating a variety of novel features, and prominent in its contributions to programming language design is its polymorphic typing system. A strongly typed language like Pascal is completely type checked at compile time, obviating the need for runtime type checking; the penalty is that code which has been written in a largely "type independent" style (stacks, trees, or even the identity function) must be repeated with only changes in type declarations. On the other hand, a naive Lisp compiler will do no compile time type checking, allowing Lisp code to be freely reused on different data types, but the price paid is run time type checking.

The ML idea of type polymorphism is a successful attempt to get part of the best of both worlds. Given an ML expression, a precisely defined *type discipline* automatically infers the functional type of the expression, or rejects the expression as untypable. The simplest example of this type polymorphism in action is the identity function. In the ML expression *let* $I = \lambda x.x$ *in* $\langle body \rangle$, the identifier I is thought to have functional type $t \rightarrow t$ for *any* type t, and separate (so-called "let-bound") instances of the identifier I in the expression $\langle body \rangle$ do not further constrain each other in terms of inferring their types. This contrasts with "lambda-bound" variables, where each instance in the body must have *identical* and not merely isomorphic type. A Lisp-like interpretation of the ML expression *let* $I = \lambda x.x$ *in* II would be as syntactic sugar for $(\lambda I.II)(\lambda x.x)$, but the ML interpretation is better thought of as $(\lambda y.y)(\lambda x.x)$, which would infer that $\lambda x.x$ was of type $t \rightarrow t$, that $\lambda y.y$ was of type $(t \rightarrow t) \rightarrow (t \rightarrow t)$, and that the entire expression was then of type $t \rightarrow t$. The Lisp-like interpretation $(\lambda I.II)(\lambda x.x)$ would be rejected by the ML type discipline, however, since the lambda-bound variable I is forced to have both type $t \rightarrow u$ (in its "function" incarnation in the body) as well as type t in its "argument" incarnation. The type discipline insists via unification that $t = t \rightarrow u$, and on the basis of this positive *occur check*, the subexpression $\lambda I.II$ is declared untypable. The fact that, of course, this example does *not* cause a type error motivates the search for more robust type disciplines.

In embedding such polymorphic type inference in a programming language, there are natural concerns that the inference mechanism be *decidable* so that the compiler can terminate, and *efficient* so that termination is within a reasonable amount of time. In his original paper, Milner made it clear that the former was true, and straightforward termination and correctness proofs (for example, [W87], which is actually a correctness proof for typing pure lambda terms, but works for ML with trivial changes) have since been published. Certainly in practice, the inference system has been efficient, which led to the belief that this efficiency was a polynomial time one, i.e., that typing an ML expression of length n could be done in time polynomial in n. Upon closer scrutiny, however, this putative folk theorem turned out instead to be unsubstantiated folk lore.

First, it was observed by several researchers, including Buneman, Kanellakis, Mitchell, and Wand, that there exist pathological ML expressions whose principal type is of vastly larger size than the original expression.

Example 1.1.

```
let $x_0 = \lambda z.z$
 in let $x_1 = \langle x_0, x_0 \rangle$
    in let $x_2 = \langle x_1, x_1 \rangle$
       in ...
          in let $x_n = \langle x_n, x_n \rangle$
             in $x_n$
```

Example 1.2.

```
let $x_1 = \lambda y.\langle y, y \rangle$
 in let $x_2 = \lambda y.x_1(x_1(y))$
    in ...
       in let $x_n = \lambda y.x_{n-1}(x_{n-1}(y))$
          in $x_n(\lambda z.z)$
```

(We use $\langle x, y \rangle$ as an abbreviation for $\lambda z.zxy$, the familiar lambda calculus implementation of pairing.)

In Example 1.1, we have a construction for ML expressions of length n which include $\Omega(2^{cn})$ distinct type variables. Even more pathological is the construction of Example 1.2, based on repeated function composition: it has a principal type which is of length $\Omega(2^{2^{cn}})$ when printed as a string, and has a representation as a directed acyclic graph with $\Omega(2^{cn})$ nodes. The author type checked the expression when $n = 5$ using Standard ML running on a Sun 3/160 workstation, a computation which consumed over 2 minutes of processor time, 60 megabytes of memory, and output 173 printed pages of the principal type until output summarily aborted. [1] This experiment places many comments about typing and software engineering in a curious light, for instance, the (by no means unusual) remark in [Pf88] that "[Types]...provide a succinct and formal documentation and thus help the programmer read, debug, and maintain his programs."

Secondly, a more sophisticated and more damning blow was struck at the folklore of efficiency of ML typing by Kanellakis and Mitchell [KM89], who showed that simply *deciding whether or not* an ML expression is indeed typable is PSPACE-hard, indicating that the difficulty of typing expressions is not merely intractible because of the size of the output, but because of complexities of a more intrinsic nature. An upper bound is given by them that the typability question (and indeed, the actual computation of the principal type) can be answered in DEXPTIME. They mention the resolution of the complexity of this decision problem as an outstanding open question, leaving speculation that its difficulty might be greater than that computable in polynomial space.

After all this bad news, some good news is in order. Unfortunately, this paper only gives more bad news: we show here that deciding ML typability for the "Core ML" language treated by Kanellakis and Mitchell is actually DEXPTIME-hard. As a consequence, deciding ML typability is complete for deterministic exponential time. We note that a simultaneous proof [KTU89a] has been announced, using altogether different methods.

This bound is of obvious relevance to the understanding the typing mechanisms of a variety of functional languages which have been built in part around the ML type discipline, e.g., Miranda, Orwell, Haskell, etc. It also provides interesting insights into problems in software engineering concerning reusable code, because let-polymorphism is precisely a mechanism for specifying that code, and ensuring that its subsequent use will not generate run-time type errors.

[1] Space limitations naturally restrict a full report on this experiment; we include a summary in Appendix A. A serious question is motivated by this little test: if a decidable type system can output 50 unreadable pages of principal type with huge computational overhead, does it really make any difference if the type system is decidable?

1.1 Lower bounds.

Lower bounds proofs relating to complexity classes generally fall into two categories: reductions from problems of known and proven difficulty, and generic reductions. For example, the PSPACE-hardness proof in [KM89] is a reduction from Quantified Boolean Formulas (QBF), already known to be complete for PSPACE. The intuition of Kanellakis and Mitchell was primarily derived from Example 1.1 above: they realized that the use of ML polymorphism essentially described in this example could also be used to simulate truth tables. Even though a truth table on n variables is of exponential size, their insight was that a short (i.e., polynomial in n) ML program could "expand" exponentially via *let*-reduction (the *let*-equivalent of β-reduction) to simulate the table.

We present in contrast a *generic* reduction: given any deterministic one-tape Turing machine M with input x running in $O(c^{|x|})$ time, we show how to construct an ML formula $\Phi_{M,x}$, such that M accepts x iff $\Phi_{M,x}$ is typable, where the length of $\Phi_{M,x}$ is polynomial in the length of the description of M and x. Since every language L in DEXPTIME has a deterministic Turing Machine M_L which can decide if $x \in L$ for input x in $O(c^{|x|})$ time, this reduction shows that the difficulty of deciding typability of ML expressions is (within a polynomial factor) as hard as deciding membership in the "hardest" languages in DEXPTIME. We note that there are languages in DEXPTIME requiring exponential time and space infinitely often, for example, deciding if a *semiregular* expression (regular operators plus intersection) over some alphabet Σ denotes Σ^* [AHU][Hu73].

The simple intuition providing the foundation of the DEXPTIME-hardness proof presented here is motivated by the above Example 1.2. The intuition is the following: note that the function x_n in the example is equivalent to the lambda term

$\lambda y.x_0^{2^{n-1}}(y)$, 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" in "DEXPTIME." It is uniquely the expressive power of ML polymorphism to succinctly express function composition which results in a proof of DEXPTIME-hardness. We conjecture that stronger lower bounds on deciding typability for extensions to the ML typing system—or, for that matter, extensions to the typed lambda calculus— can be regarded precisely in terms of this capacity for succinct function composition.

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. As we will show, all of this circuitry can be realized by lambda terms, using the Boolean gadgets of Kanellakis and Mitchell, originally proposed in their paper on the inherent sequentiality of unification [DKM84], and recycled most recently as lambda terms in their PSPACE-hardness proof. We add a Boolean "fanout" gate to their logical menagerie in the interest of facilitating our proof.

We present the proof in "bottom up" form, showing first how to encode Boolean values as lambda terms, adding Boolean logic, Turing machine state encoding, tape encoding, proceding 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, as one of my colleagues with more applied interests put it, 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.

1.2 Polymorphic unification.

We then proceed from this lower bound to a further understanding why deciding typability is so difficult, focusing our attention on the kind of type equations which must be solved (using unification) to decide whether an ML expression is typable. Given such a set of type equations, and two type variables chosen from this set, a natural question is to ask whether the variables must have the same value in the unification solution. Were this solvable efficiently (e.g., by an oracle), we show that deciding typability, and in addition the computation of the actual type, can be done in polynomial space. It is, therefore, this particular question about unification which is the bottleneck in deciding typability. It turns out that there is a natural correspondence between this question and the encoding of Boolean logic which is integral to our Turing Machine simulation.

2 The DEXPTIME-hardness bound.

We present the ML program constructed from a Turing Machine M and input x as a series of equations which are meant to be nested as a series of *let*-expressions. The program is written in "Core ML," i.e. the language defined as:

$$E ::= x \mid \lambda x.E \mid EE \mid \textit{let } x = E \textit{ in } E$$

In some instances, we also give the principal type of the expression to clarify its significance. In the ML "type hacking" which follows, we acknowledge an obvious debt to the authors of [KM89], who introduced many of the techniques; the contribution of this paper is to use them more expressively. The coding tricks used here allow types to simulate calculations by exploiting the power of polymorphism to drive the inference engine of unification, in the same spirit that Church showed how the values of lambda terms could, via β-reduction, simulate computation.

2.1 Notation. Miscellaneous combinators.

$$\begin{aligned}
I &= \lambda z.z \\
K &= \lambda x.\lambda y.x \\
Eq &= \lambda x.\lambda y.K\ x\,\lambda z.K\ (z\,x)(z\,y) : \\
&\qquad a \longrightarrow a \longrightarrow a
\end{aligned}$$

$$\begin{aligned}
&\langle \phi_1, \phi_2, \ldots, \phi_n \rangle = \\
&\quad \lambda z.z\phi_1\phi_2 \ldots \phi_n : \\
&\qquad (t_1 \longrightarrow t_2 \longrightarrow \cdots \longrightarrow t_n \longrightarrow u) \longrightarrow u \\
&pair = \\
&\quad \lambda x.\lambda y.\lambda x'.\lambda y'.\lambda z.K\ z \\
&\qquad (K\ Eq(x, x')\ Eq(y, y')) : \\
&\qquad\quad a \longrightarrow b \longrightarrow a \longrightarrow b \longrightarrow c \longrightarrow c
\end{aligned}$$

We typically write $Eq(x, y)$ instead of $Eq\ x\ y$. The importance of the Eq combinator is that for ML to correctly type $Eq(\phi, \psi)$, the ML expressions ϕ and ψ are constrained to have identical type. When ϕ and ψ are lambda-bound variables, this constraint can affect the types of other expressions: it is this phenomenon which permits us to carry out the reduction. If the constraint is impossible (i.e., causing a positive occur check), a mistyping occurs and the entire expression is rejected.

Notice that in the definition of $\langle \phi_1, \ldots, \phi_n \rangle$, we imagine the formula ϕ_i to have principal type t_i, and in the entire expression, the types t_i do not necessarily constrain each other. When an ML formula $\lambda w.Kw\langle \phi_1, \ldots, \phi_n \rangle$ is typed, it has the same principal type as the I combinator, *provided* that the type constraints introduced by the ϕ_i can be satisfied. This construct allows a transparent means of introducing constraints on types of subexpressions.

The definition of *pair* introduces the type equivalent of the Lisp **cons**. Instead of *pair* $x\,y$ we usually write $[x; y]$. When 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.

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

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

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. In the innermost *let* in our ML simulation of a Turing Machine, then, we produce a Boolean value indicating if the machine rejected its input, and apply that value to two non-unifiable arguments; the whole formula types properly iff the machine accepts. (For more details, see Section 2.12.)

2.3 Zero and One (Tape Symbols).

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

Now we define predicates telling if a cell holds a zero or a one:

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

Observe that *cell* p causes p to unify with 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 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" and "outputs," though in the relational calculus of unification they are really one and the same—it matters only which of the two you choose to constrain! The $\lambda z.Kz$ marks the end of the inputs and outputs; next comes 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."

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

The importance of this encoding scheme 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, because we have encoded whether the tape symbol is a zero in the type bound to x, and the negation in the type bound to y.

2.4 Boolean operators *and* and *or*. Fanout.

We implement these monotone Boolean operators using the gadgets introduced in [DKM84]. We add yet another gadget to implement multiple fanout, indicating why such an addition is necessary.

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

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

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

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.

Now for disjunction:

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

In typing this term, we have the constraints

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

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

An anomaly

Note that, however strong the temptation may be, these 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

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

when we would have expected the answer to be [*true*; *false*]. What happened? Imagine we have for $1 \le i \le 3$ pairs of *let*-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), $1 \le j \le 2$ encode the Boolean *or* of the first two and last two pairs. The encoding of the *or* operator enforces the following constraints:

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

Note that the type equality of x_1 and y_1 naturally forces the equality of the types of u_1 and v_1, but this forces the "argument" part of the types of x_2 and y_2 to be equal. This equality in turn forces the equality of the types of u_2 and v_2.

What has been ignored in the Boolean simulation is that the second input has multiple fanout: if we introduced constraints by typing the terms $\{x_2u_1u_2,\ y_2v_1v_2\}$ instead of typing $\{x_2u_1,\ x_2u_2,\ y_2v_1,\ y_2v_2\}$, then everything works out properly:

$$\begin{array}{ll} x_1^{a\longrightarrow f}u_1^a & y_1^{b\longrightarrow g}v_1^b \\ x_2^{a\longrightarrow c\longrightarrow h}u_1^au_2^c & y_2^{b\longrightarrow d\longrightarrow k}v_1^bv_2^d \\ x_3^{c\longrightarrow \ell}u_2^c & y_3^{d\longrightarrow m}v_2^d \end{array}$$

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.

These examples motivate 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.

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

Viewed as a dag in the style of [DKM84] (see Figure 1(a)), 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 have type $a \longrightarrow b \longrightarrow c \longrightarrow c$ ("false until proven true"); further constraints may force $a = b$. Figure 1(b) shows the type of *fanout* as a type dag.

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:

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

This definition can be used to copy tape symbols:

$$\mathit{copy\text{-}cell} \;=\; \mathit{copy}_2$$

In addition, we can use the definition of $copy_k$ to construct more than two copies of some type structure:

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

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 in the above definition, we use the "temporary" types 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.

2.5 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):

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

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

$$
\begin{aligned}
&initial\text{-}state = \\
&\quad \lambda q_1.\lambda q_2.\cdots\lambda q_n.\lambda z.Kz \\
&\qquad \langle Eq(q_1, true), Eq(q_2, false), \\
&\qquad\ Eq(q_3, false), \ldots, Eq(q_n, false)\rangle \\
&copy\text{-}state = copy_n
\end{aligned}
$$

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:

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

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.

2.6 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. Here's how: first, for lack of anything better, we define

$$nil = \lambda z.z$$

Now we use function composition to generate an exponential amount of tape using a polynomial-sized expression. Let c be a positive integer; in the following, we explicitly include the *let* syntax to emphasize the power of polymorphism needed.

$$
\begin{array}{l}
\textit{let } zero_0 = \lambda tape.[zero;\ tape] \\
\quad \textit{in let } zero_1 = \lambda tape.zero_0(zero_0\ tape) \\
\qquad \textit{in let } zero_2 = \lambda tape.zero_1(zero_1\ tape) \\
\qquad\quad \textit{in } \ldots \\
\quad \textit{in let } zero_{cn} = \lambda tape.zero_{cn-1}(zero_{cn-1}\ tape) \\
\qquad \textit{in } zero_{cn}\ nil
\end{array}
$$

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

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

where we have 2^{cn} *zeroes*. By composing the $zero_i$ functions, we can code up a list (i.e., tape) of k *zeroes* for $0 \le k \le 2^{cn}$ using an ML expression of size polynomial in n. We can then "hand code" more symbols at the end of the tape, e.g., a binary encoding for tape endmarkers.

2.7 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, $1 \le i \le n$, 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.

$$
\begin{array}{l}
\textit{next-state} = \\
\quad \lambda state.\lambda cell. \\
\qquad \lambda t_1.\lambda t_2.\cdots\lambda t_n.\lambda w.Kw \\
\qquad\quad \lambda state_1.\lambda state_2. \\
\qquad\quad \lambda cell_1.\lambda cell_2.\cdots\lambda cell_n. \\
\qquad\quad \lambda q_1^{(1)}.\lambda q_2^{(1)}.\cdots\lambda q_n^{(1)}. \\
\qquad\quad \lambda q_1^{(2)}.\lambda q_2^{(2)}.\cdots\lambda q_n^{(2)}. \\
\qquad\quad \lambda z_1.\lambda z_2.\cdots\lambda z_n. \\
\qquad\quad \lambda o_1.\lambda o_2.\cdots\lambda o_n. \\
\qquad\qquad \langle \textit{copy-state state state}_1\ state_2, \\
\qquad\qquad copy_{2,n}\ cell\ cell_1\ cell_2 \cdots cell_n, \\
\qquad\qquad state_1\ q_1^{(1)}\ q_2^{(1)} \cdots q_n^{(1)}, \\
\qquad\qquad state_2\ q_1^{(2)}\ q_2^{(2)} \cdots q_n^{(2)}, \\
\qquad\qquad cell_1\ z_1\ o_1, \\
\qquad\qquad cell_2\ z_2\ o_2, \\
\qquad\qquad\quad \ldots \\
\qquad\qquad cell_n\ z_n\ o_n, \\
\qquad\qquad Eq(t_1, \phi_1),\ Eq(t_2, \phi_2), \ldots, \\
\qquad\qquad Eq(t_n, \phi_n)\rangle
\end{array}
$$

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.

2.8 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.

$$
\begin{array}{l}
\textit{new-cell} = \\
\quad \lambda state.\lambda cell. \\
\qquad \lambda f.\lambda g.\lambda h.Kh \\
\qquad\quad \lambda state_1.\lambda state_2. \\
\qquad\quad \lambda cell_1.\lambda cell_2.\cdots\lambda cell_n. \\
\qquad\quad \lambda q_1^{(1)}.\lambda q_2^{(1)}.\cdots\lambda q_n^{(1)}.
\end{array}
$$

$$\begin{aligned}
&\lambda q_1^{(2)}.\lambda q_2^{(2)}.\cdots \lambda q_n^{(2)}.\\
&\lambda z_1.\lambda z_2.\cdots \lambda z_n.\\
&\lambda o_1.\lambda o_2.\cdots \lambda o_n.\\
&\quad \langle \textit{copy-state state state}_1\ \textit{state}_2,\\
&\quad \textit{copy}_{2,n}\ \textit{cell cell}_1\ \textit{cell}_2 \cdots \textit{cell}_n,\\
&\quad \textit{state}_1\ q_1^{(1)}\, q_2^{(1)} \cdots q_n^{(1)},\\
&\quad \textit{state}_2\ q_1^{(2)}\, q_2^{(2)} \cdots q_n^{(2)},\\
&\quad \textit{cell}_1\ z_1\, o_1,\\
&\quad \textit{cell}_2\ z_2\, o_2,\\
&\quad\quad \cdots\\
&\quad \textit{cell}_n\ z_n\, o_n,\\
&\quad Eq(f, \phi_{\text{zero?}}), Eq(g, \phi_{\text{one?}})\rangle
\end{aligned}$$

The expressions $\phi_{\text{zero?}}$ and $\phi_{\text{one?}}$ 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.

2.9 Turing Machine IDs.

We represent a Turing Machine ID by a type

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

where *state*, *left*, and *right* are type metavariables representing more complicated type structures encoding, respectively, the state of the machine (as described in Section 2.5), and *left* and *right* are 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* (see Figure 2).

$$\begin{aligned}
&\textit{initial-ID} =\\
&\quad \lambda \textit{state}.\lambda \textit{left}.\lambda \textit{right}.\lambda z.Kz\\
&\quad\quad \langle Eq(\textit{state}, \textit{initial-state}),\\
&\quad\quad Eq(\textit{left}, \textit{nil}),\\
&\quad\quad Eq(\textit{right}, \Phi)\rangle
\end{aligned}$$

where Φ is an exponential tape formula as described in Section 2.6.

2.10 Some notes on the simulation.

Let M be a Turing Machine which accepts or rejects an input $x \in \{0,1\}^*$ in $2^{c|x|}$ state transitions, for some positive integer c. We have already considered how to simulate M's state changes and its writing on the tape, but not its head movements. The reason is that it does not seem obvious (at least to this author!) how to simulate the head movements if at every state transition, the machine might move left or right. Instead, we simulate an equivalent machine having *uniform* movement of the tape head. We now clarify what the term "uniform" precisely means.

Instead of simulating M on input x, we simulate an M^* which sweeps its tape head right $2^{c|x|}$ times and then sweeps its tape head left $2^{c|x|}$ times in order to simulate *one* transition of M. It repeats this loop $2^{c|x|}$ times to simulate M's computation on x. The running time of M^* is then slower than M by an exponential factor, but is still running in exponential time.

Suppose then that M has states Q, alphabet $\Sigma = \{0,1\}$, and is running on input x. We construct another Turing Machine M'_x with states Q' and alphabet

$$\begin{aligned}
\Sigma' = \{0, 1, \$, \textit{blank}\} \cup\\
\{\langle \sigma, q, \textit{mode}\rangle \mid \sigma \in \Sigma,\ q \in Q,\\
\textit{mode} \in \{\textit{left}, OK\}\}.
\end{aligned}$$

M'_x simulates M on input x as follows:

1. M'_x writes a \$ on the tape, moves $2^{c|x|}$ tape cells to the right, and writes another \$. It then returns to the left \$ mark, writing blanks as it moves left, writes x in the left of the marked out region, again returning to the leftmost \$. We assume without loss of generality that M never moves to the left of its input $x = x_1x_2\cdots x_t$. M'_x now replaces x_1 by the symbol $\langle x_1, q_1, OK\rangle$, indicating M to be in its initial state reading the tape square with x_1 in it.

2. M'_x now begins the simulation of M's computation on x.

(i) M'_x advances its head towards the right until it encounters a tape cell labelled $\langle\sigma,q,OK\rangle$.

(ii) If $\delta_M(\sigma,q) = (q',\sigma',R)$, then M'_x replaces $\langle\sigma,q,OK\rangle$ by σ', moves 1 cell to the right, replaces the next tape symbol τ by $\langle\tau,q',OK\rangle$, and moves its read head all the way to the rightmost \$ on the tape.

(iii) If M wants to move left, on the other hand, i.e., $\delta_M(\sigma,q) = (q',\sigma',L)$, then M'_x replaces $\langle\sigma,q,OK\rangle$ by $\langle\sigma',q',\mathit{left}\rangle$, and moves to the rightmost \$.

(iv) Now for the return journey: M'_x moves its tape head left until it sees a \$ or a $\langle\sigma,q,\mathit{left}\rangle$. If it sees a $\langle\sigma,q,\mathit{left}\rangle$ symbol, let τ be the contents of the neighboring cell to the left: M'_x replaces $\langle\sigma,q,\mathit{left}\rangle$ by σ, moves one tape cell left, replaces τ by $\langle\tau,q,OK\rangle$, and then moves to the leftmost \$.

By executing (i)–(iv), M'_x simulates one state transition of M. M'_x codes accepting states of M by remembering in its finite state memory, when sweeping left from the cell labelled $\langle\sigma,q,OK\rangle$ to \$, if q is an accepting state of M, and stays in an accepting state itself during this tape traversal.

Finally, we simulate M'_x by another Turing Machine M^*, where the alphabet of M^* is just $\{0,1\}$, coding up tape symbols in Σ' by $\log_2|\Sigma'|$ bits. Deriving M^* from M'_x is tedious but straightforward; see any decent book on automata theory (e.g., [HU79]) for details. It is clear that M^* runs in exponential time and space, and reaches an accepting state iff M'_x does.

It should be noticed that the tape endmarkers \$ are really not needed, but the price paid is a complication of the ML simulation. In presenting the proof, we have taken care to present the ML code so that it looks as much as possible like a Turing Machine. An alternative is to reconfigure the code so that it computes a function mapping IDs to IDs; in this case the endmarkers could be removed.

2.11 Transition function for a "uniform" Turing Machine.

First, we code a transition of M^* moving right:

```
delta-right =
  λold-ID.
    λnew-state.λnew-left.λnew-right.λz.K z
      λstate.λleft.λright.λcell.
      λstate₁.λstate₂.λcell₁.λcell₂.
        (old-ID state left right,
        right cell new-right,
        copy-state state state₁ state₂,
        copy-cell cell cell₁ cell₂,
        Eq(new-state,
          next-state state₁ cell₁),
        Eq(new-left,
          [new-cell state₂ cell₂; left]))
```

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*). We now generate an exponential number of "move right" transitions:

```
let $\delta_0^R$ = delta-right
in let $\delta_1^R = \lambda ID.\delta_0^R(\delta_0^R\ ID)$
   in let $\delta_2^R = \lambda ID.\delta_1^R(\delta_1^R\ ID)$
      in ...
         in let $\delta_{cn}^R = \lambda ID.\delta_{cn-1}^R(\delta_{cn-1}^R\ ID)$
```

The functions *delta-left* and δ_{cn}^L can be defined similarly. Note carefully how the rightward movement of the tape head is coded into *delta-right*: the *left* list representing the tape to the left of the read head grows, and the *right* list decreases.

2.12 The simulation: Finale.

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

$$\begin{aligned}
&\textit{let } loop_0 = \lambda ID.\delta^L_{cn}(\delta^R_{cn}\, ID)\\
&\textit{in let } loop_1 = \lambda ID.loop_0(loop_0\, ID)\\
&\quad \textit{in let } loop_2 = \lambda ID.loop_1(loop_1\, ID)\\
&\qquad \textit{in } \ldots\\
&\textit{in let } loop_{cn} = \lambda ID.loop_{cn-1}(loop_{cn-1}\, ID)\\
&\quad \textit{in}\\
&\qquad \lambda state.\lambda z.K\ z\\
&\qquad\quad \langle (loop_{cn}\ \textit{initial-ID})\ state,\\
&\qquad\quad (\textit{reject? state})\ Eq\ I\rangle
\end{aligned}$$

In the above expression, we note that while *initial-ID* is indeed the initial instantaneous description of the *simulation*, it is *not* the initial configuration of the Turing Machine. The latter begins its computation by marking off an exponential amount of tape, writing the input, and returning to the leftmost endmarker; it is this state of the computation where we begin our simulation.

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, $Eq : a \longrightarrow a \longrightarrow a$ and $I : a \longrightarrow a$ will be forced to be unified, causing a mistyping.

Theorem 2.1. *Deciding whether an ML expression is typable is DEXPTIME-hard.*

2.13 Some comments on the lower bound.

The only place in the above construction where ML polymorphism is absolutely necessary is where we use exponential function composition: in constructing the exponential tape of zeroes, and in the construction of the transition function, detailed in Sections 2.6, 2.11, and 2.12. The other uses of *let* are mere notational conveniences: 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.

Observe that since the transition function can be polynomially realized by typed lambda terms, generic reductions showing PTIME-hardness and PSPACE-hardness follow easily by relaxing our use of *let* polymorphism. The simple reason that we get merely DEXPTIME-hardness (and DEXPTIME-completeness, via the upper bound in [KM89]) has nothing to do directly with Turing Machines; rather, is that we cannot compose function application any more succinctly. Since Church numerals are just function composition, we are tempted to say that ML typability is DEXPTIME-complete because we cannot count high enough, fast enough.

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. Since the lambda-calculus part of the proof encodes Turing Machines as well as simpler computing media (automata, for instance), it may well be generalizable in other ways, e.g., automata- or regular expression-based lower bounds for Girard's System F [GLT89], for example. Of course at the moment this is wishful thinking.

3 Polymorphic unification.

We have now seen that deciding typability is DEXPTIME-complete. What is it *about* deciding typability, however, that makes the problem so difficult? We now identify a certain problem concerning unification to be the root cause of the intractability of the decision problem.

A standard algorithm (whose correctness has been succinctly proven in [W87]) to decide if a lambda calculus term is typable is to use the term to generate a series of type equations over a set of type variables with a binary function symbol $\longrightarrow$. The equations are of the form $U = V$ or $U = V \longrightarrow W$. This set of equations, whose size is linear in the size of the original lambda term, is then given to a unification algorithm, which closes the set of equations over a simple *unification logic.* The closure groups type variables having the same solution into equivalence classes. These equivalence classes then can be thought of as the nodes of a special kind of directed graph, where the outdegree of every node is either 0 or 2; in the latter case, an equation $U = V \longrightarrow W$ can be interpreted as $[U] = [V] \longrightarrow [W]$, namely that $[U]$ is an equivalence class (node) with two labelled children nodes $[V]$ and $[W]$. A certain subgraph of this structure can be identified with the putative type of the original lambda term, and this term is typable iff the subgraph is acyclic.

The problem of typing Core ML expressions is virtually identical, except for the *let* polymorphism. In this case, certain subsets of type equations can be thought of as being *polymorphically reinstantiated* in the set of equations to be unified. For example, in typing *let* $x = E$ *in* B, where E is a closed lambda term, the set of type equations to be unified contains a "copy" of the type equations associated with E (reinstantiated with new type variables) for *each* free occurance of x in B. As in the case of lambda terms, the expression is typable if the graph induced by the equivalence classes created by unification is acyclic.

Given a set of type equations generated by a Core ML expression E, and two type variables U and V from the system, we may ask: in the unification solution of the system, do U and V have the same value? That is, are they in the same equivalence class? If this question can be answered, say, by an oracle requiring no computing resources, then the typability of E can be decided in PSPACE. Furthermore, if E is indeed typable, then its principal type can be output as a directed acyclic graph in polynomial space. (The careful reader will note that at this point, an earlier, failed proof that typability is in PSPACE is being recycled.) While the details of our DEXPTIME-hardness bound have been spelled out in full, we limit ourselves to a sketch of the important features of this analysis.

3.1 A canonical-form transformation.

We begin by showing how an arbitrary Core ML expression E can be transformed into another expression E' in a certain *canonical form* which preserves the principal type iff there is one. We write $|E|$ to refer to the *length* of an ML expression E, defined without loss of generality as the size of its parse tree. The length of a type is defined similarly as the size of its tree (or where relevant, dag) representation.

The canonical form we compute has the following structure:

$$
\begin{aligned}
&\mathcal{C}_n \equiv \\
&\mathit{let}\ x_0 = E_0\ \ \mathit{in} \\
&\mathit{let}\ x_1 = (\lambda\ell_1.\lambda\ell_2.\cdots\lambda\ell_{i_1}.E_1)\ x'_{1,1}\ x'_{1,2}\ \cdots\ x'_{1,i_1} \\
&\quad \mathit{in} \\
&\mathit{let}\ x_2 = (\lambda\ell_1.\lambda\ell_2.\cdots\lambda\ell_{i_2}.E_2)\ x'_{2,1}\ x'_{2,2}\ \cdots\ x'_{2,i_2} \\
&\quad \mathit{in} \\
&\qquad \cdots \\
&\mathit{let}\ x_n = (\lambda\ell_1.\lambda\ell_2.\cdots\lambda\ell_{i_n}.E_n)\ x'_{n,1}\ x'_{n,2}\ \cdots\ x'_{n,i_n} \\
&\quad \mathit{in} \\
&x_n
\end{aligned}
$$

The salient features of the canonical form are:

- Its length is polynomial in the length of the original expression.
- All the let-expressions are nested in a single chain.
- E_0 and each $\lambda\ell_1.\lambda\ell_2.\cdots\lambda\ell_{i_j}.E_j$ are all let-free and are closed lambda terms.

We now describe how this transformation can be made in polynomial time. First, we α-convert all λ- and let-bindings to be unique identifiers, so that subsequent transformation of the expression does not result in any unwanted name clashes. The initial ML expression E is then transformed using the following rule we call *let-lifting,* since each *let* is "lifted" as far "outside" of the expression as possible:

$$
\begin{aligned}
&\lambda x.\mathit{let}\ y = F\ \mathit{in}\ M \quad \Longrightarrow \\
&\qquad \mathit{let}\ y' = \lambda x.F\ \mathit{in}\ \lambda x.M[y'x/y]
\end{aligned}
$$

Repeatedly applying the above transformation to E results in an expression where no let-binding occurs in a lambda body, and preserves the principal type. The resulting expression has length at most quadratic in $|E|$.

Next, we arrange the let-bindings so that they form a uniform chain. To do so, we use the following simple type-preserving transformations until they can no longer be applied:

$$\begin{array}{ll}
\Phi\ (\mathit{let}\ x = E\ \mathit{in}\ B) & \Longrightarrow \\
\quad \mathit{let}\ x = E\ \mathit{in}\ \Phi\ B & \\
(\mathit{let}\ x = E\ \mathit{in}\ B)\ \Phi & \Longrightarrow \\
\quad \mathit{let}\ x = E\ \mathit{in}\ B\ \Phi & \\
\mathit{let}\ x = (\mathit{let}\ y = F\ \mathit{in}\ N)\ \mathit{in}\ M & \Longrightarrow \\
\quad \mathit{let}\ y = F & \\
\quad \mathit{in}\ \mathit{let}\ x = N & \\
\quad\quad \mathit{in}\ M &
\end{array}$$

The result of applying all these transformations is an expression having a single chain of let-bindings, each of which is let-free.

Theorem 3.1. *An arbitrary ML expression can be reduced to canonical form with only a quadratic increase in expression size.*

3.2 Deciding typability: the type tree.

We now use the canonical form to define an exponential-sized data structure called the *type tree*. We can then essentially dispense with the original ML expression E as well as its canonical form, and use a compact representation of the type tree to derive whether or not E is typable.

3.3 The type tree.

Notice that in the canonical form, each expression bound to x_k is defined in terms of two polymorphic copies of some set of x_j, $j < k$, imitating the structure of a tree where the nodes have different branching factors (outdegree). If the expression is typable, each x_k has a principal type derived from taking polymorphic copies of the principal types for each x_j, $j < k$, and merging them modulo a small (i.e., polynomial) set of constraints. This set of constraints is defined from the type of the closed lambda term $\lambda\ell_1.\lambda\ell_2.\cdots\lambda\ell_{i_k}.E_k$.

We use Example 1.2 in the Introduction to illustrate this point: bindings to each x_k can be written as

$$\mathit{let}\ x_k = (\lambda\ell_1.\lambda\ell_2.\lambda y.\ell_1(\ell_2 y))\ x_{k-1}\ x_{k-1}\ \ \mathit{in}\ \ldots,$$

and we may type the closed λ-term as

$$\lambda\ell_1^{b\to c}.\lambda\ell_2^{a\to b}.\lambda y^a.(\ell_1^{b\to c}(\ell_2^{a\to b}y^a)^b)^c : \\ (b\to c)\to(a\to b)\to a\to c.$$

Suppose x_{k-1} has a principal type τ, and separate copies of τ are bound to ℓ_1 and ℓ_2: the constraint that the definition of x_k places on these types is that the "output" of the type of ℓ_2 must equal the "input" of the type of ℓ_1.

Definition 3.2. *The* type tree $\mathcal{T}_n$ *of the canonical expression C_n is tree, where the root contains type equations derived from the closed lambda term*

$$\lambda\ell_1.\lambda\ell_2.\cdots\lambda\ell_{i_n}.E_n,$$

with a particular type variable X_n in the root denoting the type of x_n. The children of the root are also type trees, one for each free occurance of a let-bound variable x_j appearing in the definition of x_n. The root also contains an equation $X_{s,j} = Y_{s,j}$ where $Y_{s,j}$ is a type variable in the subtree associated with the type of the sth free occurance of x_j in the definition of x_n, and $X_{s,j}$ is the type variable associated with that free occurance in the root.

The type tree enjoys the following friendly properties:

- While the type tree has size exponential in the size of the canonical ML expression, each node in the type tree takes only polynomial space to store. Furthermore, the variables appearing in a constraint equation always appear in adjacent type tree nodes.

- Variables can be uniquely identified as to what tree node they come from, where the variable names have polynomial length encoding the path to the node, and which variable in the node is being referenced.

- The unification solution of the constraints in the tree contains the solution for the principal type of the canonical expression, if there is such a type.

The idea of the type tree is very similar to a variety of other representation schemes for describing the exponential number of type equations generated by *let* polymorphism, in particular Kanellakis' *pointer dags* [Ka] and Kfoury's *acyclic semi-unification*[Kf].

An important consequence of the above properties on which the PSPACE result (with oracle) rests is the following: even though the type tree has exponential size, it can be virtually represented in polynomial space by simply storing type equation representations of the closed lambda terms. The type equations at any node can be regenerated in polynomial time by taking the virtual "master copy" of the node, and simply renaming variables based on the path to the node.

The last point above—that the solution $\mathcal{S}$ of the constraints determines the principal type—provides insight into the core of our polynomial space algorithm. Given two variables U and V from the type tree, we want to determine in polynomial space if $U = V$ in $\mathcal{S}$, a relation we will write as $U \equiv V$ to distinguish it from the $=$ constraints in the type tree. The solution $\mathcal{S}$ defines an obvious equivalence relation among the variables which is given by $\equiv$, but we may further associate a relation $\Rightarrow$ among these equivalence classes as $[U] \Rightarrow [V]$ if $A = B \rightarrow C$ is a constraint in the type tree, $A \in [U]$, and $B \in [V]$ or $C \in [V]$. The type tree $\mathcal{T}_n$ then induces a graph $\mathcal{G}_n$, where the nodes are the equivalence classes, and the directed edges are the $\Rightarrow$ relation. The relation $\Rightarrow$ naturally encodes the idea of "type substructure," from which we have the following lemma, stating that a core ML expression is untypable iff an *occur check* returns positive.

Lemma 3.3. *The ML expression E is typable iff $\mathcal{G}_n$ is acyclic, equivalently, iff the transitive closure of $\Rightarrow$ is irreflexive.*

We now sketch some of the important features of our polynomial space solution. We are assuming that the $\equiv$ relation is computed by an oracle at no cost, from which we can compute the $\Rightarrow$ relation in polynomial space. A consequence of these assumptions is that deciding typability can also be computed in polynomial space by a simple nondeterministic algorithm. Let R be a type tree variable associated with the putative principal type of the ML expression E. Beginning with R, our algorithm simply *guesses* a *path*, i.e., a set of relations $V_0 = R \Rightarrow V_1 \Rightarrow \cdots \Rightarrow V_j$, and guesses as well to remember some V_i where $i < j$; it then checks if $V_i \equiv V_j$. If the answer is "yes," the ML expression is not typable, because a cycle was discovered. The ML expression is clearly not typable if it is possible to make a series of correct guesses leading to a "yes." A crucial reason *why* this computation is in PSPACE is that the number of V_k which must be guessed along the path is merely exponential, and thus in polynomial space we may maintain a sort of *clock* initialized to a value exponential in $|E|$, and decreased by one after each V_k is guessed; if there is a path leading to an acyclicity, it must be guessed before the clock runs to zero. By Savitch's Theorem [Sa70], we know that this algorithm can be simulated by a deterministic one with only a quadratic blowup in space, resulting in a polynomial-space algorithm.

How can the lower bound presented in Section 2 and this oracular PSPACE upper bound be understood together? First, they indicate that the bottleneck in deciding typability can be understood in terms of proving equality of type variables in the context we have detailed above. Secondly, given the oracle, deciding if our Turing Machine simulation codes an accepting computation can be decided in constant time: observe that acceptance or rejection is expressed by a Boolean variable coded as (*reject? state*). The Boolean simulation of *true* and *false* each use two type variables which may or may not be constrained to be equal; being able to test in constant time this type variable equality is analogous to examining the exponential sized Boolean "circuits" defined by ML let-polymorphism, and being allowed to test any "output wire" in constant time to see if it is a 0 or 1. In other words, the lower bound, in its simulation of Boolean values via pairs of type variables, exploits *precisely* the computational intractability of proving equivalence of type variables, the problem which the oracle was specifically intended to solve.

3.4 How to output the principal type as a directed acyclic graph

Given that we can decide if an ML expression is typable, how can we indeed output its type, and assuming that we have a $\equiv$-oracle, can this be done in polynomial space? Since the principal type can have exponential size, we clearly cannot compute and store the entire type in polynomial space; instead we ask if we can construct an output device having polynomial space which will output the principal type into an external file. We begin with the following simple observation:

Lemma 3.4. *The principal type of an ML expression E cannot be output as a string in space polynomial in $|E|$.*

Proof. Let $|E| = n$. If the output device is allowed some polynomial $p(n)$ space, it can only encode $c^{p(n)}$ states. By a simple counting argument just like the pumping lemma for finite automata, if the output device is required to generate a principal type represented as a string of length $2^{2^{cn}}$, it must either abort prematurely or loop. $\bowtie$

Instead, we show how to output the principal type as a directed acyclic graph, and in this case, such counting arguments will not work, since the dag size is at most a single exponential in $|E|$.

Definition 3.5. *A type tree variable V* has structure *if $V = P \rightarrow Q$ is a constraint in the type tree.*

Definition 3.6. *Type tree variable P is a* parent *of Q if $W = R \rightarrow S$ is a constraint in the type tree, $P \equiv W$, and either $Q \equiv R$ or $Q \equiv S$. P is* an ancestor *of R if either P is a parent of R, or P is a parent of Q and Q is an ancestor of R.*

Definition 3.7. *A* path *is a sequence $\langle U_1, \ldots, U_k \rangle$ of type variables where U_i is a parent of U_{i+1}, $1 \leq i < k$. A path $\mathcal{P}_1 = \langle U_1, \ldots, U_k \rangle$* appears to the left of *a path $\mathcal{P}_2 = \langle V_1, \ldots, V_\ell \rangle$, written $\mathcal{P}_1 \prec \mathcal{P}_2$, if $U_1 \equiv V_1$, and either $U_1 \equiv U_2 \longrightarrow V_2$, or $U_2 \equiv V_2$ and $\mathcal{P}_1 = \langle U_2, \ldots, U_k \rangle$ appears to the left of $\mathcal{P}_2 = \langle V_2, \ldots, V_\ell \rangle$. A type tree variable U* appears to the left of *type tree variable V, written $U \prec V$, if there exist leftmost paths $\mathcal{P}_1 = \langle U_1, \ldots, U_k \rangle$ and $\mathcal{P}_2 = \langle V_1, \ldots, V_\ell \rangle$ where $\mathcal{P}_1 \prec \mathcal{P}_2$, $U \equiv U_k$ and $V \equiv V_\ell$.*

Lemma 3.8. *Given type tree variables P and Q, we can determine if $P \prec Q$ in polynomial space.*

Definition 3.9. *We associate with every type tree variable V a* canonical type tree variable *$\chi(V)$ defined as follows: let $E(V) = \{W \mid W \equiv V\}$. The $W \in E(V)$ have a lexicographic ordering. Define $E_s(V) = \{W \in E(V) \mid W \text{ has structure}\}$. If $E_s(V) \neq \emptyset$, we define $\chi(V)$ to be the lexicographically minimal element of $E_s(V)$. Otherwise, we define $\chi(V)$ to be the lexicographically minimal element of $E(V)$.*

Lemma 3.10. *For any type tree variable V, we can compute $\chi(V)$ in polynomial space.*

If an ML expression E is typable, then the solution to its associated set of equational constraints contains no positive occur checks. This means precisely that the "graph encoding" of the solution (modulo $\equiv$-equivalence of variables) is acyclic. Hence we follow the following strategy: beginning with the root variable R, we depth-first search the graph following children in left to right order: if $V \equiv P = Q \rightarrow R$, we visit the node (equivalence class) associated with V, then in turn output the structures associated with Q and R. When we are about to explore such a structure, we first compute whether it has already been output, this the intent of the definition of $\prec$. Since the node of every output structure has a unique parent via the use of depth-first search, we do not need a stack to implement the tree recursion, although the space savings is at a rather prohibitive time cost. To compute the address pointers associated with nodes is easy: since each node in the graph corresponds to an equivalence class of type variables in $\mathcal{T}_n$, we choose as node pointer the canonical representative of the class (see Definition 3.9 above).

```
procedure outdag (V : type variable) :
  { only called on structures that have
    not yet been output }
  $U := \chi(V)$;
  if $U$ has structure $\longrightarrow$
    { say, $U = A \rightarrow B$ }
    output internal node $U$;
    if $\exists\, W \prec U$ where $W \Rightarrow A \longrightarrow$
      output pointer $\chi(A)$;
      if $\exists\, W \prec U$ where $W \Rightarrow B \longrightarrow$
        output pointer $\chi(B)$;
        backup($U$)
      [] else $\longrightarrow$ outdag($B$)
      fi
    [] else $\longrightarrow$ outdag($A$)
    fi
  [] else $\longrightarrow$
    { $U$ doesn't have structure;
      we've reached a leaf }
    output leaf $U$;
    backup($U$)
  fi
erudecorp;
```

Remark. When *outdag*(V) is only called on "new" structures, the leftmost parent of V is unique up to $\equiv$-isomorphism.

```
procedure backup (V : type variable) :
  if $V \equiv R \longrightarrow$ return
  [] else $\longrightarrow$
    $U := \chi(V)$;
    if $\exists$ leftmost parent $P = A \rightarrow B$ where
          $A \equiv B \equiv U \longrightarrow$
      { The second "copy" of $U$ should
        be generated by a dag pointer }
      output pointer $\chi(U)$;
      backup($P$)
    [] $\exists$ leftmost parent $P = A \rightarrow B$ where
          $A \equiv U, B \not\equiv U \longrightarrow$
      if $\exists\, W \prec B$ where $W \equiv B \longrightarrow$
        { $B$ already generated }
        output pointer $\chi(B)$;
        backup($P$)
      [] else $\longrightarrow$ outdag($B$)
      fi
    [] $\exists$ leftmost parent $P = A \rightarrow B$ where
          $A \not\equiv U, B \equiv U \longrightarrow$
      backup($P$)
    fi
  fi
erudecorp;
```

4 Conclusions.

In this paper, we have provided a simple proof that deciding ML typability is DEXPTIME-complete. 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. We identified a closely related problem in the logic of unification, namely deciding the equality of two type variables in an exponential set of type equations derived from a Core ML expression, and showed that the lower bound comes precisely from this problem. Were it solvable in polynomial space, deciding typability would be in PSPACE, and computation of the principal type would also be possible in polynomial space.

One question immediately comes to mind: given the oracle we have postulated for solving the type variable equality problem described above, is deciding typability PSPACE-hard? We conjecture this is the case: an argument based on transitive closure will probably bear this out. Can the DEXPTIME-hardness proof be reworked to get an undecidability result for the Milner/ Mycroft type system? We conjecture that it should be possible to use their FIX rule to code up an unbounded amount of blank tape, as well as an ML simulation of a Turing Machine which generates an unbounded number of IDs. This might provide some alternative intuitions to the proof of [KTU89b]. Finally, we propose using the basic Turing Machine simulation in typed lambda calculus as a tool for studying other more complex type systems.

The significance of the lower bound presented in this paper is not at all mitigated by the fact that programmers do not typically code up Turing Machines in their types. The importance of the result is that *a priori* one cannot set a *reasonable* upper bound on the amount of computation that goes on during polymorphic type checking. The Turing Machine simulation is merely an example of a particular kind of huge computation.

The question remains why everyone has believed ML polymorphic type checking to be efficient, when in the worst case it is clearly not so. Bounding the amount of nested uses of *let*, the notion of *let depth* as proposed in [KM89], is one syntactic means of assuring polynomial-time type inference, but this restriction seems arbitrary; indeed, its main virtue is that it facilitates an inductive argument in proving the upper bound. Some sort of average-case analysis would be very nice, but it is not at all clear what probabilistic distribution on programs would be convincing or realistic. Certainly distributions on trees (e.g., branching, depth) simulating the canonical form could be used to produce a result, but whether such an analysis would produce an answer of practical relevance is not obvious.

Acknowledgments. Paris Kanellakis was kind enough to send me his unpublished notes on details of the PSPACE-hardness construction sketched in his paper with John Mitchell, which allowed me to understand how the proof really worked. I would especially like to thank Paris for detailed reading of several earlier versions of this paper, some of which were truly flawed beyond belief, and for his merciless cross-examination and verification of the final DEXPTIME-hardness bound. John Mitchell also read an exposition of the lower bound, and checked many of its details by implementing them in Standard ML. I am indeed fortunate to have benefited from their advice and scrutiny, but even more fortunate to have written a proof that could practically have been checked by a computer!

I would also like to thank Dennis Kfoury for his patience and invaluable role as sounding board and critic during a feverish period in April and May of 1989, when I began thinking that the work described in [KM89] was worthy of a sequel. The opportunity to present that paper at the BU/Northeastern type theory seminar motivated my interest in this subject.

5 References.

[AS85] Harold Abelson and Gerald Sussman, Structure and Interpretation of Computer Programs. MIT Press, 1985.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

[DKM84] Cynthia Dwork, Paris Kanellakis, and John Mitchell, On the Sequential Nature of Unification. J. Logic Programming 1(1):35-50.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor, Proofs and Types. Cambridge University Press, 1989.

[HMM86] Robert Harper, David MacQueen, and Robin Milner, Standard ML. Research Report ECS-LFCS-86-2, Computer Science Department, University of Edinburgh, March 1986.

[HU79] John E. Hopcroft and Jeffrey D. Ullman, Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 1979.

[Hu73] Harry B. Hunt, The equivalence problem for regular expressions with intersection is not polynomial in tape. TR 73-156, Computer Science Department, Cornell University, 1973.

[Ka] Paris Kanellakis, personal communication.

[KM89] Paris Kanellakis and John Mitchell, Polymorphic unification and ML typing, POPL 1989.

[Kf] Dennis Kfoury, personal communication.

[KTU89a] Dennis Kfoury, Jerzy Tiuryn, and Pavel Urzyczyn, An analysis of ML typability. Preprint.

[KTU89b] Dennis Kfoury, Jerzy Tiuryn, and Pavel Urzyczyn, Undecidability of the semi-unification problem. Preprint.

[Mi78] Robin Milner, A theory of type polymorphism in programming, JCSS 17 (1978), pp. 348–375.

[Sa70] Walter Savitch, Relationship between nondeterministic and deterministic tape complexities, JCSS 4:2 (1970), pp. 177–192.

[Pf88] Frank Pfenning, Partial polymorphic type inference and higher-order unification. 1988 Conference on Lisp and Functional Programming.

[W87] Mitch Wand, A simple algorithm and proof for type inference. Fundamenta Informaticae 10 (1987).

A A pathological example

```
Standard ML of New Jersey,
Version 0.24, 22 November 1988
val it = () : unit
- fun pair x y=fn z=> z x y;
val pair = fn :
  'a -> 'b -> ('a -> 'b -> 'c) -> 'c
- let val x1=fn y=> pair y y
  in let val x2=fn y=> x1(x1(y))
     in let val x3=fn y=> x2(x2(y))
        in let val x4=fn y=> x3(x3(y))
    in let val x5=fn y=> x4(x4(y))
       in x5(fn z=> z) end end end end end;
[Major collection...
  97% used (375228/385108), 1280 msec]
  [Increasing heap to 1104k]
[Major collection...
  88% used (504800/568000), 1940 msec]
  [Increasing heap to 1480k]
[Major collection...
  [Increasing heap to 2288k]
  99% used (824308/825248), 3200 msec]
  [Increasing heap to 2416k]
[Major collection...
  [Increasing heap to 3816k]
  99% used (1313348/1314312), 5140 msec]
  [Increasing heap to 3848k]
[Major collection...
  99% used (2057024/2058152), 8400 msec]
  [Increasing heap to 6032k]
[Major collection...
  99% used (3185268/3186408), 12680 msec]
  [Increasing heap to 9336k]
[Major collection...
  99% used (4887136/4888160), 20540 msec]
  [Increasing heap to 14320k]
[Major collection...
  99% used (7394828/7395948), 31020 msec]
  [Increasing heap to 21672k]
[Major collection...
  [Increasing heap to 35016k]
  99% used (12274372/12351328), 49780 msec]
  [Increasing heap to 35968k]
[Major collection...
  99% used (19176936/19179460), 79040 msec]
  [Increasing heap to 56184k]
[Major collection...
  -48% used (28902440/28906544), 119580 msec]
  [Increasing heap to 57960k]
[Major collection...
  [Increasing heap to 58936k]
  [Increasing heap to 59192k]
  [Increasing heap to 59320k]
  -35% used (31712940/31713380), 129880 msec]
  [Increasing heap to 59352k]

val it = fn : (((((((((((((((((((((((((((((
(('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) ->
((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) ->
'c) -> 'c) -> (((((('a -> 'a) -> ('a -> 'a) ->
'b) -> 'b) -> ((('a -> 'a) -> ('a -> 'a) ->
'b) -> 'b) -> 'c) -> 'c) -> 'd) -> 'd) -> ((
(((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) ->
((('a -> 'a) -> ('a -> 'a) -> 'b) -> 'b) ->
'c) -> 'c) -> (((((('a -> 'a) -> ('a -> 'a) ->
'b) -> 'b) -> ((('a -> 'a) -> ('a -> 'a) ->
'b) -> 'b) -> 'c) -> 'c) -> 'd) -> 'd) -> 'e)
-> 'e) -> (((((((('a -> 'a) -> ('a -> 'a) ->
'b) -> 'b) -> ((('a -> 'a) -> ('a -> 'a) ->
'b) -> 'b) -> 'c) -> 'c) -> ((((('a -> 'a) ->
('a -> 'a) -> 'b) -> 'b) -> ((('a -> 'a) ->
('a -> 'a) -> 'b) -> 'b) -> 'c) -> 'c) -> 'd)
-> 'd) -> ((((((('a -> 'a) -> ('a -> 'a) ->
'b) -> 'b) -> ((('a -> 'a) -> ('a -> 'a) ->
'b) -> 'b) -> 'c) -> 'c) -> ((((('a -> 'a) ->
('a -> 'a) -> 'b) -> 'b) -> ((('a -> 'a) ->
('a -> 'a) -> 'b) -> 'b) -> 'c) -> 'c) -> 'd)
-> ... and so on for hundreds of pages! ...
```

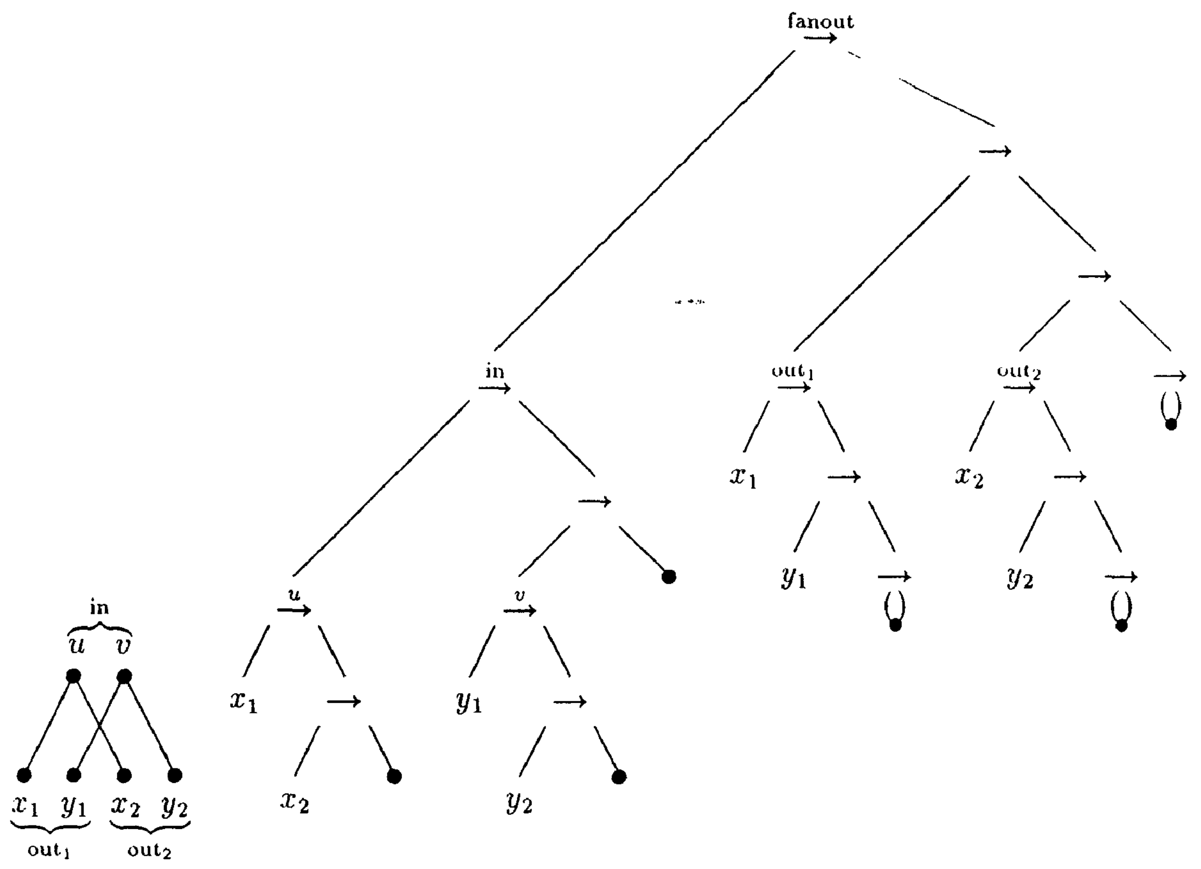

Figure 1: Implementing *fanout* as (a) a dag; (b) a type.

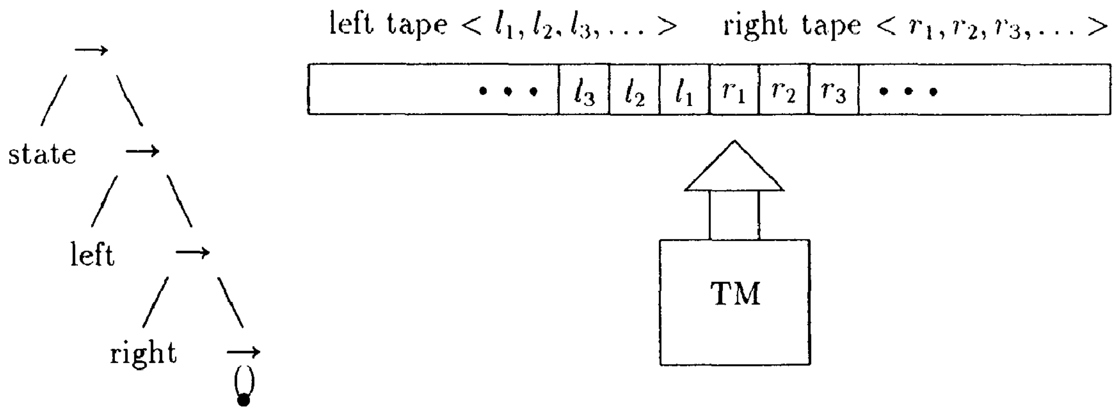
left tape $< l_1, l_2, l_3, \ldots >$
right tape $< r_1, r_2, r_3, \ldots >$
l_3 l_2 l_1 r_1 r_2 r_3
TM
state
left
right
()

Figure 2: Turing Machine IDs.