Parallel Beta Reduction Is Not Elementary Recursive

Andrea Asperti
Dipartimento di Scienze dell’Informazione, Via de Mura Anteo Zamboni, 40127 Bologna, Italy
E-mail: asperti@cs.unibo.it

and

Harry G. Mairson
Computer Science Department, Brandeis University, Waltham, Massachusetts 02454-9110
E-mail: mairson@cs.bu.edu

Received May 20, 1998; final manuscript received December 15, 1998

1. INTRODUCTION

In foundational research some two decades ago, Jean-Jacques Lévy attempted to characterize formally what an optimally efficient reduction strategy for the λ-calculus would look like, even if the technology for its implementation was at the time lacking. Lévy’s dual goals were correctness, so that such a reduction strategy does not diverge when another could produce a normal form, and optimality, so that redexes are not duplicated by a reduction, causing a redundancy in later calculation [Lévy78, Lévy80]. The relevant functional programming analogies are that call-by-name evaluation is a correct but not optimal strategy, while call-by-value evaluation is an approximation of an incorrect but optimal strategy. It is for this reason that implementers of call-by-name functional languages are interested in static program analysis (for example, strictness analysis), so that the “needless work” inherent in normal-order evaluation might somehow be controlled.

Such optimal and correct implementations were known for recursion schemas, but not ones where higher-order functions could be passed as first-class values [Vui74]. In elaborating his notion of optimal reduction, Lévy introduced a labeled variant of λ-calculus, where all subterms of an expression are annotated with labels that code the history of a computation. He proposed the idea of a redex family—redexes in a term with identical labels in the “function” position—to identify similar redexes whose...
reduction should somehow be evaluated at once (via a so-called parallel $\beta$-reduction) by any efficient scheme.

Recent research by Lamping, and independently by Kathail, has shown that there indeed exist $\lambda$-calculus evaluators satisfying Lévy’s specification [Lam90, Kat90]. Lamping introduced a beautiful graph reduction technology for sharing evaluation contexts dual to the sharing of values. His pioneering insights have been modified and improved in subsequent implementations of optimal reduction, most notably by Asperti and by Gonthier et al. [Asp94, GAL92a]. Lamping’s sharing nodes allow a single, shared representation of a redex family, and thus provide a means to implement Lévy’s notion of parallel reduction. The varied implementations of this graph reduction framework differ in the underlying bookkeeping technology used to control interactions between sharing nodes.

A fundamental and unresolved question about this sharing technology, proposed by Lamping and offered in modified from by others, is to understand the computational complexity of sharing as a function of the “real work” of $\beta$-reduction. In recent years, various papers [Asp96, LM96, LM97] have begun to address this issue. This question concerning algorithm analysis only begs more global questions that one can pose about the inherent complexity of optimal evaluation and parallel $\beta$-reduction by any implementation technology. In this paper, we take major steps toward resolving such global questions, with important algorithmic insights into the relevant graph reduction technology. Specifically, we prove that the cost of parallel $\beta$-reduction is not bounded by any Kalmár-elementary recursive function. Not only do we establish that a parallel $\beta$-step is not a unit-cost operation, but also we prove that the time complexity of implementing a sequence of $n$ parallel $\beta$-steps is not bounded as $O(2^n)$, $O(2^{2^n})$, $O(2^{2^{2^n}})$, or in general, $O(f(n))$, where $f(n)$ is any fixed stack of 2’s with an $n$ on top.

In order to make these questions and answers precise, we need to define the cost of parallel $\beta$-reduction:

**Definition 1.1.** Let $E$ be a labeled $\lambda$-term that normalizes in $n$ parallel $\beta$-steps. Let algorithm $A$ be a interpreter that normalizes the unlabeled equivalent of $\lambda$-term $E$ in $t$ time steps. We then say that $A$ implemented the $n$ parallel $\beta$-steps at cost $t$, and define the cost of a single parallel $\beta$-step in this reduction as $\frac{t}{n}$.

The point of this definition is that while algorithm $A$ need not make calculations that have anything to do with parallel reduction, we consider the time cost of parallel $\beta$-reduction, as implemented by $A$, by assigning all of the algorithmic work done—in a completely arbitrary way—to the parallel reduction steps. In any such assignment of work, at least one parallel $\beta$-step must require cost $\frac{t}{n}$.

A key insight, essential to the establishment of our nonelementary lower bound, is that any simply typed $\lambda$-term can be reduced to normal form in a number of parallel $\beta$-steps that is only linear in the length of the explicitly typed term. The point of this claim depends on the judicious use of $\eta$-expansion to control the number of parallel $\beta$-steps. Not only does $\eta$-expansion act as an accounting mechanism that allows us to see order in the graph reduction, but it also serves as a lovely sort of optimizer that exchanges the work of parallel $\beta$-reduction for the work of sharing. Our result then follows from Statman’s theorem that deciding equivalence of typed $\lambda$-terms is not elementary recursive [Sta79]. We emphasize in Statman’s theorem the generic simulation of time-bounded computation. In particular, we stress the straightforward but powerful technology of [Mai92], where a functional programming implementation of quantifier elimination for higher-order logic over a finite base type is employed to simulate arbitrary Kalmár-elementary, time-bounded computation. That the decision problem for this higher-order logic has nonelementary complexity was originally proven by Meyer [Mey74].

It is very easy to give a brief description of the proof of our lower bound, if the reader has a nodding familiarity with sharing graphs. We define the Kalmár-elementary functions $K_0(n)$ as $K_0(n) = n$, and $K_{n+1}(n) = 2^{K_n(n)}$ [Kal 43].

**Main Theorem.** Let $\ell \geq 0$ be any fixed integer. Then there exists a set of explicitly typed, closed $\lambda$-terms $E_{\ell}$ : $\text{Bool}$, where $|E_{\ell}| = O(n)$, $E_{\ell}$ normalizes in $O(|E_{\ell}|)$ parallel $\beta$-steps, and the time needed to implement the parallel $\beta$-steps, on any first-class machine model, $^3$ grows as $\Omega(K_\ell(n))$.

$^3$ A “first-class” machine model vEB90 is any computational model equal to the power of a Turing machine, modulo polynomial slowdown. For example, register machines with a logarithmic cost criterion are first-class; counter machines are not.
Proof sketch. For a fixed Turning machine $M$, and input $x$ of length $n$, consider the question, “Does $M$ accept $x$ in $K_{\ell+1}(n)$ steps?” We show how to compile this question into a succinct simply typed $\lambda$-term $E$, where the length of the explicitly typed term $E$ is exponential in $\ell$ but linear in $n$. Since $\ell + 1$ is the fixed height of the “stack of 2’s,” the exponential factor makes no asymptotic difference: it is only a constant. The term $E$ reduces to the standard $\lambda$-calculus coding for “true” if and only if the answer to the question is “yes.” The construction of the term $E$ follows from the proof of the theorems of Statman and Meyer.

We then demonstrate that the reduction to normal form requires only a linear (in the length of $E$) number of parallel $\beta$-steps. Suppose that the $|E|$ parallel $\beta$-steps could indeed be implemented at time cost $K_{\ell}(|E|)$; we would then have shown that

$$\text{DTIME}[K_{\ell+1}(|x|)] \subseteq \text{DTIME}[K_{\ell}(|E|)].$$

But the time hierarchy theorem from complexity theory (See, e.g., [Hu79]) tells us that this implied conclusion is false, since $|E|$ is polynomial in $n$; at least $K_{\ell}(n)$ time steps are necessary. Were $|E| \geq 2^n$, by contrast, the containment would not be a contradiction. The bound on the number of parallel $\beta$-steps is proven by computing the reduction of $E$ using Lamping’s algorithm. We use Lamping’s technology as a calculation device in the proof, even though the theorem concerns any implementation of optimal reduction. Since his graph reduction method is algorithmically correct, it lets us work out calculations that would be virtually impossible, and certainly intractable, in the labeled $\lambda$-calculus.

In particular, when the $\lambda$-calculus reduction of $E$ is described using Lamping’s graph reduction, we derive in only a linear number of graph reductions the sharing graph in Fig. 1. It looks just like the graphs for the $\lambda$-calculus “true” of “false,” except for the linear-sized network of sharing, bracket, croissant, and plug nodes that for technical reasons we call the blob. To know whether the graph codes “true” or codes “false,” we need to know whether the wire $a$ and plug pictured in Fig. 1 connect (respectively) to the $\lambda x$ and $\lambda y$ parameter ports, or the other way around. Deciding how these connections are made, either by graph reduction or by context semantics, must require $K_{\ell}(n)$ steps, no matter how we choose to assign the work associated with this decision problem to the individual parallel-$\beta$ steps.

That pictorial diagrams should be a mainstay of formal reasoning has not been entirely popular in theoretical computer science: consider the following charming, but ultimately withering, comments of Tony Hoare in his inaugural lecture at Oxford:

> [P]ictures actually inhibit the use of mathematics in programming, and I do not approve of them. They may be useful in first presenting a new idea, and in committing it to memory. Their role is similar to that of the picture of an apple or a zebra in a child’s alphabet book. But excessive reliance on pictures continued into later life would not be regarded as a good qualification for one seeking a career as a professional author. It is equally inappropriate for a professional programmer. Confucius is often quoted as saying that a picture is worth then thousand words—so please never draw one that isn’t. [Hoa85]

But as Richard Feynman and Julian Schwinger showed in the history of quantum electrodynamics, a picture can indeed be worth then thousand equations: Feynman was able to diagrammatically work out calculations that seemed interminable by more formal means (see, e.g., [Dys79, Sch94]). Lamping’s
graphs give a similarly important insight into the inherent process of optimal reduction that transcends the particularities of his implementation technology.

From a logical and proof-theoretical perspective, the main result shows that in implementation of optimal evaluation, almost no work is done by cut elimination, while perversely, almost all the work is carried out by structural rules. In fact, a reinterpretation of the main theorem gives bounds on the complexity of cut elimination in multiplicative-exponential linear logic (MELL), and in particular, an understanding of the “linear logic without boxes” formalism in [GAL92b], since that logic is a close analog of simply typed $\lambda$-calculus.

The lower bound characterizes the work that must be done by any technology which implements Lévy’s notion of optimal reduction. However, in the significant case of Lamping’s solution, we can also make some important remarks addressing how work done by $\beta$-reduction is translated into equivalent work carried out by his sharing and bookkeeping nodes. In essence, the result shows that the real computational work being done by Lamping’s algorithm is not accomplished by the parallel $\beta$-step, but rather by the ancillary methodology which facilitates that operation. In particular, we identify the computational paradigms of superposition of values\(^4\) and of higher-order sharing, appealing to compelling analogies with quantum mechanics and SIMD-parallelism.

None of this means that optimal evaluation is a bad idea, or that it is inherently inefficient. The computation theorist’s idea of generic simulation is comparable with the classical physicist’s idea of work; just as real work requires an expenditure of energy, generic simulation requires an unavoidable commitment of computational resources. What we learn from the analysis of this paper is that the parallel $\beta$-step is not even remotely an atomic operation. Yet the proposed implementation technology remains a leading candidate for correct evaluation without duplicating work—what we have gained is a far more precise appreciation for what that work is. In particular, sharing is real work. We believe that the graph reduction algorithm is still parsimonious in its careful handling of sharing, even if parallel $\beta$-steps are necessarily resource-intensive.

The material in this paper is entirely self-contained and from first principles. We present in Section 2 an explanation of the graph reduction technique and in Section 3 a description of the $\eta$-expansion method. Section 4 shows how to describe succinctly generic elementary-time bounded computation in higher-order logic, and how to compile expressions in this logic into short typed $\lambda$-expressions, where “optimal” is defined in a precise but entirely per-

4 In choosing this terminology, we recall how Shannon named his information-theoretic measure of uncertainty:

“Information” seemed to him to be a good candidate as a name, but “Information” was already badly overworked. Shannon said he sought the advice of John von Neumann, whose response was direct. “You should call it “entropy” and for two reasons: first, the function is already in use in thermodynamics under that name; second, and more importantly, most people don’t know what entropy really is, and if you use the word “entropy” in an argument you will win every time!” [Tri78].

2. LAMPING’S GRAPH REDUCTION TECHNIQUE

The graph reduction algorithm due to Lamping was designed for the optimal handling of shared redexes in the evaluation of $\lambda$-expressions, where “optimal” is defined in a precise but entirely persuasive sense. While the formal definition of optimal reduction is fairly technical, its basic idea is not too difficult to communicate. As a motivating example, consider reduction of the untyped term $(\lambda f. \lambda z. z(f \ M)(f \ N))(\lambda x. F \ x)$, which $\beta$-reduces to $\lambda z. z(F[M/x])(F[N/x])$. Similar redexes in the residual copies of $F$ are now duplicated, where they in fact ought to be shared.

Lamping’s idea was to decompose this sharing into two components: the sharing of the single value $F$ by two different evaluation contexts (each corresponding to an occurrence of $F$), while the “hole” $x$ in $F[x]$ is simultaneously shared (or, in the interest of duality, unshared) by two different values $M$ and $N$. The crucial point is that we cannot just share expression that represent values, but must also share contexts that represent continuations. Moreover, sharing a context—that is, a term with one or more holes inside—requires unsharing when existing through a hole.
Lamping’s graph reduction algorithm consists of a set of local rewrite rules that can be classified naturally into two groups. First, we have rules involving application, abstraction, and sharing, which are responsible for implementing $\beta$-reduction and duplication; we shall call this group of rules the abstract system. Second, we have rules involving the control nodes called bracket and croissant, which are required for the correct implementation of the first set of rules.

In particular, when two sharing nodes interact, they either duplicate or annihilate each other, depending on local information that is effectively computed by the control nodes. We can then think of the first set of rules as requiring an oracle to discriminate the correct interaction rule between a pair of sharing nodes; the second set of rules can be viewed as an implementation of this oracle.

We use Lamping’s graph reduction technique only as a convenient and correct tool for reasoning about optimal reduction, and the details of the implementation of the “oracle” by the control nodes do not play any essential role in the proof. As a consequence, we shall not discuss the details of the control node interactions; the interested reader may consult [AG98].

2.1. Initial Translation

We shall not berate the reader with the detailed definition of the simply typed $\lambda$-calculus. Recall simply that types $T$, variables $V$, and terms $E$ are generated from the following inductive definitions:

\[
T ::= o | T \rightarrow T \\
V ::= v_1 | v_2 | v_3 | \cdots \\
E ::= E | EE | \lambda V : T . E
\]

Type $o$ is called the base type. We suppress parentheses as much as possible, associating the arrow constructor to the right, and (dually) application to the left. Thus type $\alpha \rightarrow \beta \rightarrow \gamma$ means $\alpha \rightarrow (\beta \rightarrow \gamma)$, and term $EFG$ means $(EF)G$.

Associated with each well-typed $\lambda$-term is a fixed type for each of its subterms, including the term itself; we thus write $E : \tau$ to mean well-typed term $E$ has type $\tau$. Well-typed terms are determined according to the following simple rules: every occurrence of the same free variable has the same type; every occurrence of a bound variable $x$ has the same type $\alpha$ as declared in the binding $\lambda x : \alpha \ldots$; every subterm $\lambda x : \alpha . B$ has type $\alpha \rightarrow \beta$, when $\beta$ is the type of $B$; and $EF$ has type $\beta$, when $E$ has type $\alpha \rightarrow \beta$ and $F$ has type $\alpha$.

In the optimal graph reduction technique, a $\lambda$-term is initially represented by a graph that is virtually identical to its abstract syntax tree. Unlike ordinary graph reduction, however, we introduce two variations: an explicit node for sharing, and an explicit connection between variable occurrences (represented by wires) and the $\lambda$-nodes that represent their respective binders. Since we work in a typed setting, we shall label each edge of the graph with a suitable type; for brevity, we shall usually use the notation $\beta^\alpha$ instead of $\alpha \rightarrow \beta$. We emphasize that these type annotations have no operational role in reduction of the $\lambda$-term; like the names of the planets, they are merely an invariant that provides information to the analyst.

For instance, the graph in Fig. 5a is the initial representation of the $\lambda$-term $M = (\bar{2}_2 \circ o \bar{2}_a)$, where $\bar{2}_a = \lambda s : \alpha \rightarrow \alpha . \lambda z : \alpha . s(sz) : (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$. The type $(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ is the type of Church numerals (or iterators) for $\alpha$ and is usually denoted by $N_\alpha$.

The triangular node, referred to as a sharing node or a fan node, is used to express the sharing between different occurrences of a same variable. All variables are connected to their respective binders; we shall always represent this connection on the left of the connection to the body. Multiple occurrences of the same variable are represented by a binary tree of fan nodes, where these fan nodes are the internal nodes of the tree, and the occurrences are the external nodes (leaves) of the tree. As a consequences, only a single wire edge, at the root of the binary tree, connects the $\lambda$-node representing a binding to the wires representing the variable occurrences.

Each node in the graph (apply, $\lambda$, and fan) has exactly three distinguished ports where it can be connected to other ports. One of these ports (depicted with an arrow in Fig. 2) is called principal: it is the only port where the node may interact with other nodes; see the interaction rules below. The other ports are called auxiliary ports. The sharing graph and the corresponding term it represents are
well typed if and only if for each node \( n \) in the graph, the types of the edges connected to \( n \) satisfy the constraints imposed in Fig. 2, whose interpretation should be clear.

We call the type of a node the type of its principal port. Instead of typing edges, we could equivalently type each port of a node—adding, however, a suitable polarity. Our approach has been adopted for essentially typographical reasons.

2.2. Reduction

We shall now illustrate the main ideas of Lamping’s optimal graph reduction technique by showing how it works on the example term \((\overline{2} \circ \overline{2})\). As we shall see, the rules governing interaction of sharing nodes remain unresolved; this ambiguity is resolved by the “oracle” implemented by control nodes. But as the details of the implementation of the oracle are unnecessary for our analysis, we omit them.

Since Lamping’s algorithm consists of a set of local graph rewriting rules, at any given stage of the computation, we may have several reducible configurations in the graph. The choice of the next rule to apply is then made nondeterministically. This ambiguity does not change the normal form of graphs, since the graph rewriting system is an interaction net in Lafont’s sense [Laf90] and satisfies a one-step diamond property. This property implies not only confluence, but also that if a term has a normal form, all of its normalizing derivations have the same length. In the following exegesis, we choose the next rule according to didactic criteria and occasionally for graphical convenience.

The most important of the graph rewriting rules is \( \beta \)-reduction, where \((\lambda x. M)N\) is replaced by \( M[N/x] \); see Fig. 3. In graph reduction, the substitution of a term \( N \) for the bound variable \( x \) is simulated by connecting the variable wire to the graph representation of \( N \). The contractum \( M[N/x] \) is then represented by the (instantiated) graph representation of the “procedure body” \( M \) of the function. These reductions have nothing to do with the structure of \( M \) and \( N \), only with the local wiring connections between them; as a consequence, the simulation of \( \beta \)-reduction can be expressed by the completely local graph rewriting rule in Fig. 4. It is important to note that the reduction preserves the correct typing of the graph—that is, it connects edges with equal type. This property is a straightforward but essential consequence of all rewriting rules in Lamping’s algorithm.

By firing the outermost \( \beta \)-redex in \((\overline{2} \circ \overline{2})\), we derive the graph in Fig. 5b. The next redex involves a shared \( \lambda \)-expression in the function position. In ordinary graph reduction, the entire representation of the function would be duplicated. In contrast, the optimal graph reduction technique proceeds in a lazy fashion, duplicating the external \( \lambda \)-node, but still sharing its body; see Fig. 5c. Since the binder has been duplicated, thus allowing the sharing of the function body between two contexts, we dually introduce another sharing node on the edge leading from the binder to the variable, in order to “unshare” the arguments to the function. The sharing (fan-in) and the unsharing (fan-out) come in pairs. Although

![FIG. 2. Nodes, ports, and their types.](image1)

![FIG. 3. \( \beta \)-Reduction.](image2)
there is no operational distinction between a fan-in and a fan-out, their intuitive semantics is quite different; in particular, one should recognize that a fan-out is always supposed to be paired with some fan-in the graph, delimiting its scope and annihilating its sharing effect. The determination of correct pairing between such sharing nodes is a crucial aspect of the optimal graph reduction technique, solved by the control nodes—or alternatively, if we speak only of the abstract algorithm, the "oracle." We remark only that the naive oracle of labeling each sharing node in the initial graph with a unique identifier, to be copied when the sharing node is copied, fails to distinguish the appropriate pairs of nodes (Fig. 6; see [Lam90]).

Again, since the body of the function \( \lambda x. M \) does not play any role in this reduction, it can be formally expressed as a local interaction between a fan and a \( \lambda \), as described in Fig. 7. Note that the type of fan nodes (that is, the type of their principal port) may only decrease by the firing of this rule.

Next, by applying the fan-\( \lambda \) interaction rule, we get the graph in Fig. 5c. Two \( \beta \)-redexes have thus been created: by firing each of them, we derive the graph in Fig. 8a. Then we fire the fan-\( \lambda \) rule, obtaining the graph in Fig. 8b. There are no more \( \beta \)-redexes in this graph, as well as no fan-\( \lambda \) interactions, but we must proceed in the duplication process with considerable caution nonetheless. In particular, the graph rewriting rule shown in Fig. 9 is strictly forbidden in an optimal evaluation, although it is in some sense semantically correct. The intuition proscribing the rule should be clear: since the shared application could be involved in some redex, its duplication would imply a double execution of the redex, violating optimal.

The only other possible interaction is between the two fans inside the dotted region. This situation highlights another crucial point of the optimal graph reduction technique. Because these two fans in Fig. 8b are note "paired"—the fan-in is a residual of the shared variable of \( \overline{2}_o \), while the fan-out is a residual of the shared variable of \( \overline{2}_{o-\lambda} \), in the process of duplicating \( \overline{2}_o \)—they must duplicate each other, according to the rule in Fig. 10b. Note that the type of fan-nodes is preserved by this interaction. Now (see Fig. 8c), we have a fan-out in front of the function-port of the application. In this case, we can apply the rule in Fig. 11. Intuitively, this rule is correct from the point of view of optimal sharing, since such a configuration already implies the existence of two unsharable redexes for the application. As in the case of fan-\( \lambda \) interaction, the type of the sharing node strictly decreases.
FIG. 6. The duplication of abstractions.

FIG. 7. Fan-\(\lambda\) interaction.

FIG. 8. Graph reduction of \((\alpha_2 \rightarrow_\alpha \lambda_2)\).

FIG. 9. Non-optimal duplication of the application.

FIG. 10. Fan-annihilation rule.
By firing this rule twice, we derive the graphs in Fig. 12b where we have three pairs of fans interacting with each other. In these cases, all fans are paired: they each belong to the same “duplication process” that has now been locally completed. In this case, the obvious rule is to annihilate the paired fans, according to Fig. 10a.

By three applications of the fan-annihilation rule, we derive the graph in Fig. 12c. The three last steps are, respectively, a fan-\(\lambda\) rule and two \(\beta\)-reductions; see Fig. 13. The graph in Fig. 13c is in normal form with respect to Lamping’s algorithm; we urge the reader to understand why this graph represents the Church integer \(4\_4\) for four. The simplest way to see why is to complete the duplication process by applying the “forbidden” rule of duplication of the application. Alternatively, try to read-back the term by travelling inside its structure, with the following proviso: exit a fan-out node on the same side (\(\ast\) or \(\circ\)) that the paired fan-in node was entered.

3. THE \(\eta\)-EXPANSION METHOD

Given a simply typed \(\lambda\)-term \(E\), we show how to construct a variant \(E'\) that is \(\beta\eta\)-equivalent to \(E\), derived by introducing \(\eta\)-expansions of bound variables in \(E\). The size of \(E'\), and the size of the initial sharing graph that represents \(E'\), are larger than \(E\) by only a small constant factor. Moreover, the number of parallel \(\beta\)-steps needed to normalize \(E'\) is linearly bounded by its size. As a consequence, we demonstrate that the normal form of any simply typed \(\lambda\)-term can be derived in a linear number of parallel \(\beta\)-steps. In order to make these calculations more precise, we need to define what we mean by the size of types, \(\lambda\)-terms, and graphs:

**Definition 3.1.** We define the size of a simple type by structural induction:

\[
\|\alpha\| = 1
\]

\[
\|\alpha \rightarrow \beta\| = 1 + \|\alpha\| + \|\beta\|.
\]
Similarly, we inductively define the size of simply typed term as

$$|x| = ||\sigma|| \quad \text{if } x \text{ has type } \sigma$$

$$|\lambda x : \sigma.E| = 1 + |E|$$

$$|E F| = 1 + |E| + |F|.$$  

The number $|E|$ simply counts 1 for each lambda and apply in $E$, plus $||\sigma||$ for each variable of type $\sigma$. Finally, we refer to the size $\|G\|$ of a sharing graph $G$ as the number of its nodes.

The only unusual feature of this definition is that the size of a variable is given by the size of its type. Had we instead used the more usual definitions $|x| = 1$ and $|\lambda x : \sigma.E| = 1 + ||\sigma|| + |E|$, the size of terms would be polynomially smaller, but only by a quadratic factor.

The bound we prove on the number of parallel reductions depends essentially on controlling the duplication of lambda- and apply-nodes by sharing nodes. When a sharing node has type $\alpha \rightarrow \beta$ and faces the principal port of either a lambda-node or an apply-node, duplication creates two sharing nodes, of types $\alpha$ and $\beta$ respectively. If the value being shared by a node is the base type $o$, then that sharing node cannot interact with a lambda or apply-node, since the principal ports of those nodes cannot sit on wires that are at base type—they are functions.

As a consequence, each sharing node has a capacity for self-reproduction that is bounded by the size of the type of the value being shared. The idea of introducing $\eta$-expansion is to force a node sharing a value $x$ of type $\sigma$ to the base type $o$, by making that node duplicate components of the graph coding the $\eta$-expansion of $x$. This technique leads to an efficient accounting mechanism which bounds the duplication of lambda and apply-nodes in a graph reduction, and hence bounds the number of parallel beta-steps. In addition, it serves as a lovely optimization method, where parallel beta-reduction is simulated by the interaction between sharing nodes only.

**Definition 3.2.** Let $x$ be a variable of type $\sigma$. The $\eta$-expansion $\eta_\sigma(x)$ of $x$ is the typed lambda-term inductively defined on $\sigma$ as

$$\eta_\sigma(x) = x$$

$$\eta_{\alpha_1 \rightarrow \cdots \rightarrow \alpha_k \rightarrow o}(x) = \lambda y_1 : \alpha_1 \cdot \cdots \cdot \lambda y_k : \alpha_k . x(\eta_{\alpha_1}(y_1)) \cdots (\eta_{\alpha_k}(y_k)).$$

In the graph representation, each $\eta$-expanded variable is coded by a subgraph with two distinguished wires that we respectively call the positive entry and the negative entry of the variable (see Fig. 14). When the variable is of base type $o$, this subgraph is just a wire. The graph representing an $\eta$-expanded term has the following nice properties:
**Lemma 3.3.** If $G$ is the initial sharing graph representing $\eta_\sigma(x)$, then $\|G\| \leq 2\|\sigma\|$.

**Proof.** By induction on $\sigma$. Notice that if $\sigma \equiv \alpha_1 \to \alpha_2 \to \cdots \alpha_k \to \sigma$, then $\|G\| = 2k + \sum_{1 \leq i \leq k} \|G_{\alpha_i}\|$, where $G_{\alpha_i}$ represents $\eta_\sigma(y_i)$.

**Lemma 3.4.** Let $x$ be a variable of type $\sigma$, and let $G$ be the graph representing $\eta_\sigma(x)$. Then in an optimal reduction, any sharing of $G$ at the positive or negative entry results in a residual graph $\Delta(x)$, where all copies of the sharing node are at base type and $\|\Delta(x)\| \leq 3\|\sigma\|$.

**Proof.** The proof is a simple induction on $\sigma$. If $\sigma = o$, the conclusion is immediate and trivial—no reductions are possible, because $G$ is just a single wire.

Suppose $\sigma = \alpha_1 \to \cdots \to \alpha_n \to \sigma$. Then the graph $G$ coding $\eta_\sigma(x)$ has the structure depicted in Fig. 14a, with $n\lambda$-nodes and $n$ apply-nodes. If a sharing node is placed at the positive entry, that node will duplicate the $n\lambda$-nodes; copies of the sharing node will move to the auxiliary (noninteraction) port of the top apply-node, which has base type $o$, and to the negative entries of the subgraphs $G_i$ representing $\eta_\sigma(y_i)$; see Fig. 15a. Dually, if a sharing node is placed at the negative entry, that node will duplicate the $n$ apply-nodes; copies of the sharing node will move to the auxiliary port of the top $\lambda$-node, which has base type $o$, and to the positive entries of the subgraphs $G_i$ representing $\eta_\sigma(y_i)$; see Fig. 15b. The lemma follows by induction on the $\alpha_i$.

**Example 3.5.** The graph in Fig. 16 shows $\Delta(\eta_{(o \to o) \to o \to o}(x))$, the duplication of the graph for $\eta_{(o \to o) \to o \to o}(x)$ by a single sharing node at the positive entry.

The previous lemma may be easily generalized to an arbitrary tree network of sharing nodes, or equivalently the $t$-fold multiplexors of Guerrini [Gue96]. In this case, the duplicated “skeleton” of the sharing graph representing $\eta_\sigma(x)$ is replicated for each instance of its use by the $t$ leaves of the multiplexor.

**Corollary 3.6.** Let $\Delta'(x)$ be the residual graph that results from the sharing of the graph representing $\eta_\sigma(x)$ by a binary tree of $t$ sharing nodes. Then $\|\Delta'(x)\| \leq (2 + t)\|\sigma\|$.

In the initial sharing graph coding a $\lambda$-term $\lambda x : \sigma . E$, multiple references to the bound variable are represented by exactly such a tree network of sharing nodes. This tree is connected to the (auxiliary) parameter port of the $\lambda$-node that represents the binding. Because this parameter port is not a primary port, no interaction is possible between the sharing nodes and the $\lambda$-node; the sharing are “stuck” until the $\lambda$-node is annihilated in a parallel $\beta$-reduction. We seek to control the possible node replication that could result from such a reduction, by forcing the $\lambda$-term $\lambda x : \sigma . E$ to be applied to $\eta_\sigma(x')$. We may then

---

5 We chose the notation $\Delta(x)$ to remind the reader that the graph is defined by propagating the sharing nodes (hence the $\Delta$) to base type.
conclude that the duplication caused by a sharing node is proportional to a fixed function of the size of the initial term and is not affected by the size of intermediate terms of the reduction sequence. In other words, the duplication is amenable to control via static analysis of the initial term. These intuitions are clarified in the following definition.

**Definition 3.7.** Let $M$ be a simply typed $\lambda$-term. The optimal root $\text{or}(M)$ of $M$ is derived by replacing every subterm of the form $\lambda x : \sigma. E$ with $\lambda x : \sigma. (\lambda x : \sigma. E)(\eta_\sigma(x'))$, where $\sigma \neq o$ and $x$ occurs more than once in $E$. We refer to the new $\beta$-redexes introduced by this transformation as preliminary redexes.

It should be clear that the transformation is applied at most once to any subterm $\lambda x : \sigma. E$. Since $\text{or}(M)$ is obtained by $M$ by means of $\eta$ and $\beta$-expansions, we also know that $M = \beta_\sigma\text{or}(M)$. The transformation can be understood as duplicating, once and for all, the “skeleton” of this term that described all of its possible uses. The relevant information about these uses is provided unambiguously by the type $\sigma$.
Definition 3.8. We define $\Delta(M)$ to be the sharing graph obtained from or($M$) by reducing all of the preliminary redexes, and propagating all sharing nodes to the base type.

We emphasize the following crucial property of $\Delta(M)$:

**Lemma 3.9.** All sharing nodes in $\Delta(M)$ have atomic types.

*Proof.* After the $\beta$-reduction of all preliminary redexes, all sharing nodes are positioned to interact at the positive entry of the subgraphs representing $\eta$-expanded variables. Following these essentially structural reductions, the sharing nodes duplicate these subgraphs. Lemma 3.4 ensures that all such sharing nodes can propagate to base type.  

**Definition 3.10.** A bound variable is trivial in a typed $\lambda$-term if it is at base typed or occurs at most once. A typed $\lambda$-term is trivial if all of its bound variables are trivial.

**Lemma 3.11.** Let $F \equiv \lambda x : \sigma. E$ be a nontrivial term where $E$ is trivial. Then $\|\Delta(F)\| \leq 2|F|$.  

*Proof.* Let $G$ be the sharing graph coding or($F$) $\equiv \lambda x' : \sigma. F(\eta_\sigma(x'))$, and let $G'$ be driven by graph reduction of the outermost $\beta$-redex, representing the term $\lambda x' : \sigma. E[\eta_\sigma(x')/x]$ (see Fig. 15).

Assume $x$ occurs $t$ times in $E$, where $G_E$ is the sharing graph representing $E$, so that the sharing of $\eta_\sigma(x')$ is represented in $G'$ by a tree of sharing nodes with $t$ leaves. Since $\|G_E\|$ counts only the number of occurrences of $\lambda$ and apply in $E$, it should be clear that $\|G_E\| \leq |E| - t||\sigma||$.

The construct $\Delta(F)$ from $G'$, we need only propagate the sharing nodes into the graph representation of $\eta_\sigma(x')$, generating $\Delta^{-1}(x')$ with size at most $(1 + t)||\sigma||$, by Corollary 3.6. Then the size of $\Delta(F)$ is

$$\|\Delta(F)\| \leq 1 + (|E| - t||\sigma||) + (1 + t)||\sigma|| = 1 + |E| + ||\sigma|| = |F| + ||\sigma||.$$

However, $||\sigma|| \leq |F|$, since $||\sigma||$ just counts the contribution of one occurrence of $x$ in $F$; we then conclude $\|\Delta(F)\| \leq 2|F|$.

**Theorem 3.12.** Let $F$ be a simply-typed $\lambda$-term. Then $\|\Delta(F)\| \leq 2|F|$.  

*Proof.* The proof is just a generalization of the previous lemma. We construct $\Delta(F)$ by insertion of the same preliminary redexes, reduction, and propagation of sharing nodes through the graphs representing $\eta_\sigma(x)$ over variables $x$ of type $\sigma$. Assume without loss of generality that the name of each bound variable is unique; then by Lemma 3.11,

$$\|\Delta(F)\| \leq \left|F| - \sum_{x \text{ nontrivial}} \mu(x)||\sigma||\right| + \left(\sum_{x \text{ nontrivial}} (1 + \mu(x)||\sigma||)\right) \leq |F| + \left(\sum_{x \text{ nontrivial}} ||\sigma||\right),$$

where $\mu(x)$ is the number of occurrences of $x$ in $F$. When $\mu(x) \geq 2$, only $\mu(x) - 1$ sharing nodes are needed in the initial graph representation. Again, it is clear that $\sum_x ||\sigma|| \leq |F|$, so that $\|\Delta(F)\| \leq 2|F|$.

The reader may be bothered by this “linear” bound, which depends on the definition of the size function $|F|$, where the occurrence of an $x$ of type $\sigma$ contributes $||\sigma||$ to the sum. Suppose we had instead chosen the definition of size as

$$\Lambda(x) = 1$$

$$\Lambda(\lambda x : \sigma. E) = 1 + ||\sigma|| + \Lambda(E)$$

$$\Lambda(E F) = 1 + \Lambda(E) + \Lambda(F),$$

so that $\Lambda(E)$ is the length of the explicity typed term. Since it is not hard to show that $|M| \leq \Lambda(M)^2$, we would derive instead $\|\Delta(F)\| \leq 2\Lambda(F)^2$ as the statement of the previous theorem. The significance of either inequality is that $\|\Delta(F)\|$ is only polynomial in $|F|$, which is a good enough bound to derive our more important results.

As an obvious consequence of Lemma 3.9, we have the following important observation.
The total number of $\beta$-reductions (and thus of families) in the graph normalization of $\Delta(M)$ cannot exceed its initial size.

Proof. Since all sharing nodes have atomic types in $\Delta(M)$, they cannot interact with abstractions or applications. As a consequence, no new application or $\lambda$-node can be created during the reduction. Since $\beta$-reduction can only make the graph smaller via annihilation of complementary $\lambda$- and apply-nodes, the total number of $\beta$-reductions is bounded by the initial size of the graph $\Delta(M)$. □

If $\Delta(M)$ is considered as the representation of a logical proof via the Curry–Howard analogy, with the caveat that some sense is made of “fan-out,” it gives a very interesting “normal” form, where a logical rules are “below” the structural ones. This provides for some computational amusement: one can immediately and easily remove all the logical cuts. The rest of the computation is merely structural—the annihilation or duplication of sharing nodes.

A fundamental consequence of the bound on the number of redex families in $\text{or}(M)$ is its strong normalization, as well as the strong normalization of $M$. This observation is a trivial corollary of a result due to Lévy ([Lévy 78], Theorem 4.4.6):

Theorem 3.14. Let $\sum$ be any finite (possibly parallel) reduction of a term $M$. Then any reduction $\sum'$ relative\(^6\) to $\sum$ is terminating.

Since all redexes created along any reduction of $\text{or}(M)$ eventually belong to some of its families, any reduction strategy is terminating.

Theorem 3.15. The simply typed $\lambda$-calculus is strongly normalizing.

Even more, the bound on the reduction is, if one counts parallel $\beta$-reduction of families, merely linear in the length of the initial term.

4. SIMULATING GENERIC ELEMENTARY-TIME BOUNDED COMPUTATION

Now that we know that the normal form of a simply typed $\lambda$-term can be computed in a linear number of parallel $\beta$-steps, our goal is to construct a generic reduction from the largest time hierarchy we can manage via a logspace reduction.\(^3\) For example, if we can simulate deterministic computation in $\text{DTIME}[2^n]$ (deterministic exponential time) in the simply typed $\lambda$-calculus, where the initial $\lambda$-term corresponding to a computation has length bounded by a fixed polynomial in $n$, we may then conclude that the parallel $\beta$-reduction cannot be unit cost. The reason is simple: were a parallel $\beta$-step implemented in unit cost, we would have shown that $\text{DTIME}[2^n]$ since an exponential-time computation on an input of size $n$ can be compiled into a short (i.e., with length polynomial in $n$) typed $\lambda$-term, and that term normalizes in a polynomial number of parallel $\beta$-step to a Boolean value indicating acceptance or rejection of the input. We would then have simulated an exponential-time computation in polynomial time, and this conclusion contradicts the time hierarchy theorem (see, e.g., [HU79], Section 12.3), which asserts that polynomial time is properly contained in exponential time.\(^8\)

In fact, for any integer $\ell \geq 0$, we can construct generic reductions of this kind for $\text{DTIME}[K_\ell(n)]$. The consequence is that the cost of a sequence of $n$ parallel $\beta$-steps is not bounded by any of the Kalmárelementary recursive functions $K_\ell(n)$. Observe that were the cost contained in $O(K_\ell(n))$, then by simulating arbitrary computations in $\text{DTIME}[K_{\ell+1}(n)]$, and requiring only polynomially many $\beta$-steps (via the $\eta$-expansion method) at cost $O(K_\ell(p(n)))$, for some polynomial $p$, we would have shown that $\text{DTIME}[K_{\ell+1}(n)]$ is contained in $\text{DTIME}[K_\ell(p(n))]$, which is again contradicted by the time hierarchy theorem.

\(^6\) $\sum'$ is relative to $\sum$ if all redexes in $\sum'$ are in the same family of some redex in $\sum$ (see [Lévy78], Definition 4.4.1, p. 64).

\(^7\) This reduction is just a compiler that takes an arbitrary Turning machine $M$ running in some time bound $t(n)$ on an input $x$ of size $n$ and produces a typed $\lambda$-term $e : \text{bool}$ such that $e$ reduces to the term coding $\text{true}$ iff $M$ accepts $x$ in $t(n)$ steps. The “logspace” means that the compiler has only $O(\log n)$ bits of internal memory to carry out the compilation, ensuring that the output has length polynomial in $n$.

\(^8\) Readers familiar with the diagonalization technique from the proof of undecidability of the Halting Problem should recognize that in exponential time, one can diagonalize over every polynomial time computation.
4.1. Deciding Truth of Formulas in Higher-Order Logic

Rather than code directly in the simply typed $\lambda$-calculus—not always a pretty sight—we use an equivalently powerful logical intermediate language: higher-order logic over a finite base type. The process of quantifier elimination for this logic is easily simulated by primitive recursive iteration in the simply typed $\lambda$-calculus. The complexity of deciding truth for this logic was originally analyzed by Meyer, and the relation of this analysis to types $\lambda$-calculus was pointed out by Statman [Mey74, Sta79]. In what follows, we present a slight modification of the simple, short proofs of these theorems, found in [Mai92], as well as enlarging the logical language somewhat in the interest of readability.

Let $\mathcal{D}_0 = \{0, 1, \ldots, b - 1\}$ with a total ordering $<_0$, and let $\mathcal{D}_{i+1} = powerset(\mathcal{D}_i)$. We analyze the complexity of deciding the truth of formulas with quantification over elements of $\mathcal{D}_i$, for any $i \geq 0$, using a naive interpretation. Let $x', y', z'$ be variables allowed to range over the elements of $\mathcal{D}_i$; we define the prime formulas as $a <_0 b, a = b, a \in y^1$ (where $a$ and $b$ are either constants representing elements of $\mathcal{D}_0$ or variables ranging over elements of $\mathcal{D}_0$), and $x' \in y^{i+1}$. Now consider a formula $\Phi$ built up out of prime formulas, the usual logical connectives $\lor, \land, \rightarrow, \neg$, and the quantifiers $\forall$ and $\exists$: is $\Phi$ true under the usual interpretation?

4.2. Primitive Recursive Iteration in Higher Types

The truth of formulas in higher-order logic can be decided by compiling them into short types $\lambda$-terms and by using the power of primitive recursion to realize quantifier elimination and to decide the truth of prime formulas. The primitive recursion is implemented using list iteration, a straightforward programming technique that we now describe briefly. Let $\{x_1, x_2, \ldots, x_l\}$ be a set of $\lambda$-terms, each of first-order type $\alpha$; then

$$L \equiv \lambda c : \alpha \rightarrow \tau \rightarrow \tau. \lambda n : \tau. c\!\cdot\!\tau\!\cdot\!x_1\!\cdot\!\cdots\!\cdot\!x_l\!\cdot\!n \ldots$$

is a $\lambda$-term of type $(\alpha \rightarrow \tau \rightarrow \tau) \rightarrow \tau \rightarrow \tau$, for any type $\tau$. We abbreviate this list construction as $[x_1, x_2, \ldots, x_l]$; observe that the variables $c$ and $n$ abstract over the list constructors $\text{cons}$ and $\text{nil}$. In the simply types $\lambda$-calculus, list iteration can be used to implement primitive recursion. For example, given $\lambda$-terms $\text{suc}$ and $\text{0}$ for zero and successor on Church numerals, the length of a list of terms of type $\alpha$ can be computed by

$$\text{length} \equiv \lambda L : (\alpha \rightarrow \text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \rightarrow \text{Int}.L(\lambda x : \alpha.\text{suc}) \text{0},$$

where $\text{Int} \equiv (v \rightarrow v) \rightarrow v \rightarrow v$, and $\tau$ is set to $\text{Int}$ in the above definition of $L$.

List iteration is ideal for realizing quantifier elimination: imagine that we code $\mathcal{D}_i$ as a $\lambda$-term $\Phi_i$ which lists all elements of $\mathcal{D}_i$, each coded appropriately as a $\lambda$-term of type $\Delta_j$, and we have coded a Boolean function $\Phi$ as a $\lambda$-term $\Phi_i$ of type $\Delta_j \rightarrow \text{Bool}$. Then the truth of $\forall x'. \Phi(x')$ can be coded as the $\lambda$-term $\Phi_i(\lambda x' : \Delta_j, \text{AND}(\Phi(x'))) \text{true}$, and the truth of $\exists x'. \Phi(x')$ can be coded as the $\lambda$-term $\Phi_i(\lambda x' : \Delta_j, \text{OR}(\Phi(x'))) \text{false}$, where $\text{AND}$, $\text{OR}$, and $\text{true}$, and $\text{false}$ are $\lambda$-terms coding up Boolean logic. Observe, for example, that the latter reduces to $\text{OR}(\Phi e_1), \text{OR}(\Phi e_2) \ldots (\text{OR}(\Phi e_k)\text{false}) \ldots$, where $e_j$ is a $\lambda$-term coding the $j$th element of $\mathcal{D}_i$, $1 \leq j \leq k = |\mathcal{D}_i|$. As we will see, the prime formulas can also be simulated using list iteration.

4.3. Coding Elements of the Type Hierarchy

Let $\text{Bool} \equiv o \rightarrow o \rightarrow o$, and define the Boolean values and logical connectives via their usual Church codings:

$$\text{true} \equiv \lambda t : o.\lambda f : o. t : \text{Bool}$$
$$\text{false} \equiv \lambda t : o.\lambda f : o. f : \text{Bool}$$
$$\text{AND} \equiv \lambda p : \text{Bool}.\lambda q : \text{Bool}.\lambda t : o.\lambda f : o. p\!\cdot\!q\!\cdot\!f \!\rightarrow\! \text{Bool} \rightarrow \text{Bool}$$
$$\text{OR} \equiv \lambda p : \text{Bool}.\lambda q : \text{Bool}.\lambda t : o.\lambda f : o. p\!\cdot\!q\!\cdot\!f \!\rightarrow\! \text{Bool} \rightarrow \text{Bool}$$

In fact, we need to type $\Phi_i$, so that it can output an Boolean value. Since $\Phi_i$ is a list used for primitive recursive iteration, its output type $\tau$ needs to be set to a type $\text{Bool}$ coding Boolean values.
\[ NOT \equiv \lambda p : \text{Bool}. \lambda t : o. \lambda f : o. \text{pft} : \text{Bool} \rightarrow \text{Bool} \]
\[ \text{if} \equiv \lambda p : \text{Bool}, \lambda q : \text{Bool}, \text{or}(\lambda p) q : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}. \]

Projection functions represent the elements of \( D_0 \), where \( i \) is coded by \( e_i \):
\[ e_i \equiv \lambda x_0 : o. \lambda x_1 : o. \ldots \lambda x_{b-1} : o. \cdot x_j : N \equiv o \rightarrow o \rightarrow \cdots \rightarrow o \rightarrow o. \]

Using these projection functions, the total order \( <_0 \) on \( D_0 \) is defined by tabulation:
\[ \text{less}_0 \equiv \lambda x : N. \lambda y : N. \]
\[ \lambda t : o. \lambda f : o. \]
\[ x(yfft \cdots t)(yfft \cdots t) \cdots (yff \cdots f) \]
\[ \quad : N \rightarrow N \rightarrow \text{Bool}. \]

Then the set \( D_0 \) can be represented as the list \( D_0 \) of its coded elements:
\[ D_0 \equiv \lambda c : N \rightarrow t \rightarrow t. \lambda n : t. c e_0(c e_1 \cdots (c e_{b-1} n)) : (N \rightarrow t \rightarrow t) \rightarrow t \rightarrow t. \]

We abbreviate the type of \( D_0 \) as \( \Delta_0 \); in general, let \( \Delta_{k+1} \equiv \Delta_0^k \), where for any type \( \alpha \), we define \( \alpha^* \equiv (\alpha \rightarrow t \rightarrow t) \rightarrow t \rightarrow t \), and \( \Delta \equiv N \). Recall that the order of a type is a measure of its higher-order functionality, where \( \text{order}(o) = 0 \) and \( \text{order}(\alpha \rightarrow \beta) = \max(1 + \text{order}(\alpha), \text{order}(\beta)) \); thus the order of \( \Delta_0[t := \text{Bool}] \) is 3, and \( \Delta_j[t := \text{Bool}] \) is \( 2j + 3 \).

Next, for each integer \( t > 0 \), we define an explicitly typed \( \lambda \)-term \( D_t \) representing \( D_t \) as a list of (recursively defined codings of) all subsets of elements of \( D_{t-1} \) in the type hierarchy. To do so, we must introduce an explicit powerset construction to build succinct terms coding these lists. First, we define a term \( \text{double} \) where, given an element \( x : \alpha \) and a list \( \ell : \alpha^* \) of lists of elements of type \( \alpha \), \( \text{double} \) appends \( \ell \) to a list derived from adding \( x \) to each list in \( \ell \). For example, when \( \alpha \equiv \text{Bool}, \text{double} \) \( [\text{false}, \text{false}, \text{true}, [], [], \text{true}] \) reduces to \([[\text{false}]], [[\text{false}, \text{true}], [], [], \text{true}]]\).
\[ \text{double} \equiv \lambda x : \alpha. \lambda \ell : (\alpha^* \rightarrow t \rightarrow t) \rightarrow t \rightarrow t. \]
\[ \lambda c : \alpha^* \rightarrow t \rightarrow t. \lambda n : t. \]
\[ \ell(\lambda c' : \alpha^* \cdot c (\lambda c' : \alpha \rightarrow t \rightarrow t. \lambda n' : t. c' x (e c' n'))))(\ell c n) \]
\[ \text{double} : \alpha \rightarrow \alpha^{**} \rightarrow \alpha^{**}. \]

Notice that if a \( \lambda \)-term \( A^* \) coding a list of \( \lambda \)-terms of type \( \alpha \) has type \( \alpha^* \equiv (\alpha \rightarrow t \rightarrow t) \rightarrow t \rightarrow t \) for any \( t \), then \( A^* \) also has type \( \alpha^*[t := \lambda^{**}] \equiv (\alpha \rightarrow \alpha^{**} \rightarrow \alpha^{**}) \rightarrow \alpha^{**} \rightarrow \alpha^{**} \). We may then define
\[ \text{powerset} \equiv \lambda A^* : (\alpha \rightarrow \alpha^{**} \rightarrow \alpha^{**}) \rightarrow \alpha^{**} \rightarrow \alpha^{**}. \]
\[ A^* \text{double}(\lambda c : \alpha^* \rightarrow t \rightarrow t. \lambda n : t. c (\lambda c' : \alpha \rightarrow t \rightarrow t. \lambda n' : t. n'))n) \]
\[ \text{powerset} : ((\alpha \rightarrow \alpha^{**} \rightarrow \alpha^{**}) \rightarrow \alpha^{**} \rightarrow \alpha^{**}) \rightarrow \alpha^{**}. \]

The function of \( \text{powerset} \) on lists is like that of \( \text{exponentiation} \) realized via doubling on Church numerals, since Church numerals are just lists having length but containing no data.

Now we can succinctly define terms coding levels of the type hierarchy:
\[ D_1 \equiv \text{powerset} D_0 : \Delta_1 \equiv (\Delta_0 \rightarrow t \rightarrow t) \rightarrow t \rightarrow t \]
\[ D_2 \equiv \text{powerset} D_1 : \Delta_2 \equiv (\Delta_1 \rightarrow t \rightarrow t) \rightarrow t \rightarrow t \]
\[ \cdots \]
\[ D_{k+1} \equiv \text{powerset} D_k : \Delta_{k+1} \equiv (\Delta_k \rightarrow t \rightarrow t) \rightarrow t \rightarrow t. \]
In the definition of $D_{i+1}$, we use $D_i$ as an iterator, where $D_i$ is typed as $\Delta_i[\tau := \Delta_{i+1}] = (\Delta_{i-1} \rightarrow \Delta_{i+1}) \rightarrow \Delta_{i+1} \rightarrow \Delta_{i+1}$.

We now give some bounds on the growth rate of $|D_i|$ as a function of $i$. Note that the pure $\lambda$-term with type information erased grows only as $\Theta(t + b^2)$, and the length of its normal form grows as $\Theta(K(b))$. The explicitly typed term $D_i$ has a greater length due to its type information; this added verbosity is crucial when we transform terms using the $\eta$-expansion method of the previous section, and so we pay particular attention to bounding its size.

**Theorem 4.1.** If $D_i : \Delta_i$ as in the above typing, $|D_i| = O((b + 2t + 1)!)$.

**Proof.** While the bound on the growth rate may seem enormous, we will only be interested in cases where $t$ is a constant and $b = \lceil \frac{\log n}{\log \log n} \rceil$, in which case $|D_i| = O(n)$. To compute a series of upper bounds $|D_i| \leq d_i$, we define a recurrence where $d_{i+1}$ is computed from $d_i$. Recall $D_i \equiv \text{powerset} D_{i-1}$, and that to give $D_i$ the type $\Delta_i$, we take the typing of $D_{i-1}$ and assign $\tau$ the type $D_i$.

If $D_{i-1} : \Delta_{i-1}[\tau := \Delta_i]$, then this typing of $\Delta_{i-1}$ satisfies $|D_{i-1}| \leq d_{i-1} \| \Delta_i \|$. Similarly, if

$$\text{powerset} : ((\Delta_{i-2} \rightarrow \Delta_{i-1} \rightarrow \Delta_i) \rightarrow \Delta_i) \rightarrow \Delta_i,$$

then as $\| \Delta_{i-2} \| \leq \| \Delta_i \|$, we know $|\text{powerset}| \leq p \| \Delta_i \|$ for a fixed constant $p$ independent of $i$. We then conclude

$$|D_i| = |\text{powerset} D_{i-1}| \leq 1 + p \| \Delta_i \| + d_{i-1} \| \Delta_i \| \leq (1 + p) \| \Delta_i \| + d_{i-1} \| \Delta_i \| \leq 2 \| \Delta_i \| d_{i-1} = d_i,$$

assuming $1 + p \leq d_{i-1}$, which we will justify shortly. Unwinding the recurrence, we see that

$$|D_i| \leq 2 \| \Delta_i \| \cdot 2 \| \Delta_{i-1} \| \cdots \cdot 2 \| \Delta_0 \| \cdot d_0 \leq 2^t d_0 \prod_{1 \leq j \leq i} \| \Delta_j \|.$$

A simple calculation shows that $\| \Delta_j \| = 8j + 2b + 7 \leq 8(b + j + 1)$; thus

$$|D_i| \leq 16^t d_0 \prod_{1 \leq j \leq t} (b + j + 1) = 16^t d_0 \frac{(b + t + 1)!}{(b + t)!},$$

so that for large $t$, we have the crude but entirely satisfactory bound,

$$|D_i| \leq d_0 (b + 2t + 1)!$$

We need only justify the assumption $p + 1 \leq d_{i-1}$. Note simply that we may choose, as a basis for the recursive definition of the $d_i$, the value $d_1 = \max \{p + 1, \| \Delta_0 \| \}$, and that as $d_i = 2 \| \Delta_i \| d_{i-1}$, we know that $d_{i-1} \leq d_i$, and hence $p + 1 \leq d_i$, for all $t \geq 1$.\hfill \blacksquare

It is a little surprising that $|D_i|$ grows like the factorial of $t$, rather than merely exponential in $t$. Since powerset is the set-theoretic version of exponential, the $D_i$ are the set-theoretic versions of Church numerals. In the closely related iterated exponential $2^{2^{\cdots^2}}$, one sees only an exponential increase in the size of the explicitly typed $\lambda$-term as a function of the number of occurrences of $2$, due to the iterated substitution of the form $\tau := \tau \rightarrow \tau$. This constant-sized substitution multiplies the size of a term by a constant factor, and the cumulative effect is exponential. By contrast, in the substitutions $\tau := \Delta_0$, used to type iterated powerset, each such substitution multiplies the size of the $\lambda$-term by a factor of $t$, and the cumulative effect is like the factorial.

### 4.4. Coding Set Theory in the $D_i$.

There is a natural idea of *equality* between elements of $D_i$; when these elements are themselves sets, we can also define the idea of *subset* and of *element* of a set. It is then straightforward to realize the
prime formulas of higher-order logic by using list iteration. For each integer \( t > 0 \), we define terms eq\(_t\), subset\(_t\), and member\(_t\). When \( t = 0 \), we define only

\[
eq_0 = \lambda x : \mathbb{N}. \lambda y : \mathbb{N}. \lambda t : \mathbb{O}. f y t \cdot f y t \cdot f y t \cdot f y t \cdot f y t
\]

as a basis, just as we coded \( \prec_0 \) over \( D_0 \) via tabulation.

For the inductive case of the definitions, we define \( \Delta^\text{Bool}_j = \Delta_j[\tau : = \text{Bool}] \), which is the type of an iterator with Boolean output, and define

\[
\begin{align*}
\text{member}_{t+1} & = \lambda x : \Delta^\text{Bool}_t. \lambda y : \Delta^\text{Bool}_t. \lambda z : \Delta^\text{Bool}_t. \lambda l : \Delta^\text{Bool}_t \to \text{Bool}. \\
& \quad (\Delta^\text{Bool}_t \to \Delta^\text{Bool}_t \to \text{Bool}) (\Delta^\text{Bool}_t \to \Delta^\text{Bool}_t \to \text{Bool}) \text{false} \\
\text{subset}_{t+1} & = \lambda x : \Delta^\text{Bool}_t. \lambda y : \Delta^\text{Bool}_t. \lambda z : \Delta^\text{Bool}_t. \lambda l : \Delta^\text{Bool}_t \to \text{Bool}. \\
& \quad (\Delta^\text{Bool}_t \to \Delta^\text{Bool}_t \to \text{Bool}) (\Delta^\text{Bool}_t \to \Delta^\text{Bool}_t \to \text{Bool}) \text{true} \\
\text{eq}_{t+1} & = \lambda x : \Delta^\text{Bool}_t. \lambda y : \Delta^\text{Bool}_t. \lambda z : \Delta^\text{Bool}_t. \lambda l : \Delta^\text{Bool}_t \to \text{Bool}. \\
& \quad (\Delta^\text{Bool}_t \to \Delta^\text{Bool}_t \to \text{Bool}) (\Delta^\text{Bool}_t \to \Delta^\text{Bool}_t \to \text{Bool}) \text{true}
\end{align*}
\]

**Lemma 4.2.** The \( \lambda \)-terms defining member\(_t\), subset\(_t\), and eq\(_t\), each have length \( \Theta(t^2 + b^3) \); with type information erased, they have length \( \Theta(t + b^3) \).

**Proof.** Observe how each of these terms depends on only one such other term, and how the trick in the definition of eq\(_t+1\) eliminates the exponential blowup that would result from writing subset\(_t+1\) twice. The quadratic difference between the respective lengths of typed and untyped terms is explained by the fact that \( \| \Delta^\text{Bool}_t \| = \Theta(t + b) \). Notice as well that coding these operations in the typed \( \lambda \)-calculus produces much shorter terms than those that result from coding power-set.

The above definitions give a typed \( \lambda \)-calculus interpretation to all the logical formulas in our higher-order logic, in the spirit of their standard logical meaning. The elements of \( D_0 \) are interpreted by projection functions, with the prime formulas interpreted by the codings above. The logical connectives, interpreted by their Church codings, take arguments of type \( \text{Bool} \), producing terms of type \( \text{Bool} \). Quantifier elimination, as described earlier, interprets \( \forall x : \Phi(x) \) as the iterated conjunction \( \beta_1(\lambda x : \Delta^\text{Bool}_t. \Phi(x)) \), where \( \beta \) is the interpretation of \( \Phi \); the complementary interpretation of \( \exists x : \Phi(x) \) is the iterated disjunction \( \gamma_1(\lambda x : \Delta^\text{Bool}_t. \neg\Phi(x)) \).

When compiling a formula in higher-order logic into a typed \( \lambda \)-term, we can use \( \lambda \)-abstraction to ensure that the code for any of the above definitions appears only once, as in an ML-like monomorphic let \( x = E \) in \( B \), interpreted as \( (\lambda x : B. E) \). We consequently have the following conclusion:

**Theorem 4.3.** A formula \( \Theta \) in higher-order logic over the finite base type \( D_0 = \{ 0, 1, \ldots, b-1 \} \) is true if and only if its types \( \lambda \)-calculus interpretation \( \hat{\Phi} : \text{Bool} \) is \( \beta\eta \)-equivalent to true \( \equiv \lambda t : O. \lambda f : O. t \text{Bool}. \) Moreover, if \( \Phi \) only quantifies over universes \( D_i \), for \( i \leq t \), then \( \hat{\Phi} \) has order\(^{10} \) at most \( 2t + 6 \), and if we also write \( |\Phi| \) to denote the length of logic formula \( \Phi \), then \( |\hat{\Phi}| = O(|\Phi|(b + 2t + 1)) \).

\(^{10}\)By this bound, we mean that in a type derivation of \( \hat{\Phi} \), for any subderivation \( \{ t_i : \tau_i \vdash E : \tau \} \), the maximum order of any \( \tau_i \) is \( 2t + 5 \), and the maximum order of \( E \) is \( 2t + 6 \).
4.5. EXPRESSING TIME-BOUNDED COMPUTATION IN HIGHER-ORDER LOGIC

The higher-order logic provides a nice metalanguage to describe types \(\lambda\)-terms whose intuition is even more inescrutable: the logic lets us talk about the extensional proposition, where the \(\lambda\)-term embodies the intensional disposition. It only remains for us to use this logic to express Turing machine instantaneous descriptions (IDs) of sufficient size, and to compute the transitive closure of the transition relation between such IDs. The logical coding problems that have to be solved are straightforward, but fun: the comendium of tricks reads like the catalog of primitive recursive functions in Gödel’s first incompleteness theorem. Our goal is to define the formula

\[
\exists u \exists v. \text{INITIAL}(u) \land \partial^*(u, v) \land \text{FINAL}(v),
\]

where \text{INITIAL}(u) expresses that \(u\) codes or desired initial configuration, \text{FINAL}(v) expresses that \(v\) codes our final, accepting configuration, and \(\partial^*(u, v)\) expresses that the existence of a Turing machine computation from the initial to final configurations, such that the number of steps in the computation is no more than \(K_f(n)\) on an input to the Turing machine of size \(n\). The variables \(u, v, w\) must be chosen to range over a universe \(D\) large enough to code computations.

The formula that we construct in higher-order logic expressing an elementary-time computation will also have the important property of being succinct—it will not be much longer than the length of the input to the Turing machine computation. To make this notion of succinctness precise, we define the length of formulas in higher-order logic:

**Definition 4.4.** Let \(v\) be a function mapping variables to the size of their unique representations, where that representation is arbitrary but fixed. This function ensures that a formula with \(n\) variables needs length at least \(\tilde{\Omega}(n \log n)\). We then define the length \(|\Phi|\) of a formula \(\Phi\) in higher-order logic by structural induction:

\[
\begin{align*}
|\{x^k\}| &= k + v(x) \\
|\{x^k \in y^{k+1}\}| &= 2k + 2 \\
|\{x^k = y^k\}| &= 2k + 1 \\
|\neg \Phi| &= 1 + |\Phi| \\
|\Phi \circ \Psi| &= 1 + |\Phi| + |\Psi| \\
&\quad \circ \in \{\land, \lor, \rightarrow\} \\
|Qx^k.\Phi| &= k + 1|\Phi| \\
Q &\in \{\forall, \exists\}.
\end{align*}
\]

The formula we construct to describe an elementary-time Turing machine computation on input \(x\) will have length \(O(|x|)\).

4.5.1. Topo Contents

First, assume that the Turing machine we simulate has tape alphabet \(\{0, 1\}\), so that the tape contents is just a big binary number. Since \(|D_{t+1}| = 2^{|D_t}|\), it is easy to show that each element \(x^{t+1} \in D_{t+1}\) can be thought of as such a large integer, where the elements of \(x^{t+1}\) are just the bit positions set to 1 in its binary encoding.

If for a suitably large \(k\), \(x^{k+1}\) can be thought of as the tape contents, then a variable \(h^k\) can code the position of the tape head. To write \(h^k \in x^{k+1}\) expresses that the Turing machine is reading a 1, and \(h^k \notin x^{k+1}\) means the Turing machine is reading a 0. To move the tape head to the left or right corresponds to computing successor for this representation of bounded integers: if \(h^k\) and \(h^k\) code head positions in successive IDs, then \(\text{succ}_0(h^k, h^k)\) expresses that the head moved one position to the right, and \(\text{succ}_1(h^k, h^k)\) expresses that the head moved one position to the left.
We define \( \text{succ}_{t+1} \) by first extending the order \( <_{t+1} \) on \( \mathbb{D}_t \) to elements of higher universes:

\[
x^{t+1} <_{t+1} y^{t+1} \iff \exists z^t : z^t \in y^{t+1} \land z^t \notin x^{t+1} \\
\land \forall w^t : w^t <_t w' \to (w' \in x^{t+1} \iff w' \in y^{t+1}).
\]

(Translation: \( x < y \) if the \( t \)th bit in \( y \) is 1, but in \( x \) is 0, and the bits of higher order than \( z \) are identical in \( x \) and \( y \).) Successor is then defined for each \( \mathbb{D}_t \) as

\[
\text{succ}_t(x^t, y^t) \equiv x^t <_t y^t \land \forall z^t : x^t <_t z^t \to (y^t <_t z^t \lor z^t = z^t).
\]

Now we fix the cardinality of \( \mathbb{D}_0 = \{0, 1, \ldots, b - 1\} \) to facilitate the Turing machine simulation. Since \( |\mathbb{D}_t| = K_t(b) \), we can think of an element of \( \mathbb{D}_t \) as coding \( K_{t-1}(b) \) bits (or binary tape cells) of information. We henceforth take \( b = \lceil \log \log n \rceil \); note that for \( n \geq 4 \), we have \( 2^{2^n} > n \). As a consequence, \( |\mathbb{D}_{t+2}| \geq K_t(n) \) for large \( n \), so that an element of \( \mathbb{D}_{t+3} \) codes \( K_t(n) \) bits of information—enough to code the contents of a Turing machine tape with that many cells.

### 4.5.2. Constants, Pairing, and Projection

To code tape contents, head position, and finite state into a single set, we need pairing and projection relations. A standard set-theoretic definition of pairing would code \((u', v')\) simply as the set \( \{[u'], [u', v']\} \in \mathbb{D}_{t+2} \). Because our simulation uses only a small number of pairs, and our universes \( \mathbb{D}_t \) are actually a little bigger than we need, we use their “extra room” to code pairs in a more contrived way, but one which minimizes the size of the logical expression and its related typed \( \lambda \)-term.

In any universe \( \mathbb{D}_t \), \( t > 0 \), we can define zero and a small number of constants:

\[
\text{zero}_t(s^t) \equiv \forall b^{t-1} b^{t-1} \notin s^t \\
\text{con}_t^1(s^t) \equiv \exists z^t : \text{zero}_t(z^t) \land \text{succ}_t(z^t, s^t) \\
\text{con}_t^{m+1}(s^t) \equiv \exists z^t : \text{con}_t^m(z^t) \land \text{succ}_t(z^t, s^t).
\]

Since integers are essentially represented as binary numbers, we can define a “shift” operator (multiply by 2), a bitwise union, and a “spreading” function that maps the set denoting \( \Sigma b_t 2^t \) to that denoting \( \Sigma b_t 4^t \) by inserting a 0 in between each of the bits \( b_t \):

\[
\text{shift}_t(x^t, y^t) \equiv \exists z^{t-1} : \text{zero}_{t-1}(z^{t-1}) \land z^{t-1} \notin y^t \land \forall a^{t-1}. \forall b^{t-1} \land \text{succ}_{t-1}(a^{t-1}, b^{t-1}) \\
\land (a^{t-1} \in x^t \iff b^{t-1} \in y^t)
\]

\[
\text{union}_t(x^t, y^t, z^t) \equiv \forall w^{t-1} \land w^{t-1} \in z^t \iff (w^{t-1} \in x^t \lor w^{t-1} \in y^t)
\]

\[
\text{spread}_t(x^t, y^t) \equiv \forall z^{t-1} \cdot z^{t-1} \in x^t \iff (\exists w^{t-1} \cdot \text{shift}_{t-1}(z^{t-1}, w^{t-1}) \land w^{t-1} \in y^t).
\]

Now we can define a pairing function as

\[
\text{pair}_t(a^t, b^t, p^t) \equiv \exists u^t \cdot \exists v^t : \exists w^t \cdot \text{spread}_t(a^t, u^t) \land \text{spread}_t(b^t, v^t) \\
\land \text{shift}_t(u^t, w^t) \land \text{union}_t(u^t, w^t, p^t).
\]

In the last definition, \( a \) and \( b \) are paired to form \( p \), all in the same universe \( \mathbb{D}_t \). Given that an element of this universe only has \( K_{t-1}(b) \) bits, we can only iterate pairing\(^{11}\) a finite number of times; otherwise we run out of “logical memory.” Recall, however, the definition \( b = \lceil \log \log n \rceil \); it is not hard to show that for any fixed integer \( c > 0 \), for sufficiently large \( n \) (depending on \( c \)), \( K_{j-2}(b) \geq c K_t(n) \). This essentially means that our logical formula specifying the behavior of a Turing machine can contain

\(^{11}\) By this, we mean expressions such as \( \text{pair}(a_1, a_2, p_1) \land \text{pair}(p_1, a_3, p_2) \land \text{pair}(p_2, a_4, p_3) \land \cdots \land \text{pair}(p_{r-2}, a_{r-1}, p_r) \). Note that if each of the \( a_i \) have \( m \) bits of information, then \( p_i \) needs \( j 2^i \) bits.
any fixed number of pairs, and asymptotically, the “word length” in $D_{e+1}$ is big enough to support that pairing.

In the interest of clarity, for $Q \in \{\forall, \exists\}$ we use the following abbreviations:

$$Q(a', b').\Phi(a', b') \equiv Q p'.\forall a'.\forall b'.\Phi(a', b', p') \rightarrow \Phi(a', b'),$$

$$\langle a', b' \rangle \in s^{t+1} \equiv \exists p'.\Phi(a', b', p') \land p' \in s^{t+1}.$$  

4.5.3. Simulating a Turing Machine

A small but suitably large $D_{e+1}$ can code the finite states of the Turing machine as an integer. To code the tape, recall that an element of $D_{e+1}$ is big enough to code $K_2(n)$ tape cells, so let $k = e + 2$; if $u^{k+1}$, $h^k$, $s^k$, $t^k$ respectively code the tape contents, head position, finite state control, and time (i.e., the number of transition steps) for a Turing machine. We can represent all this information in a variable $z^{k+1}$ via the following tupling relation, which codes a Turing machine ID:

$$\text{ID}(u^{k+1}, h^k, s^k, t^k, z^{k+1}) \equiv \exists v^k.\exists w^k.\exists u^{k+1}.
\text{pair}(h^k, s^k, v^k) \land \text{pair}(v^k, t^k, w^k)
\land \text{single}_{k}(u^k, w^{k+1}) \land \text{pair}(w^{k+1}, u^{k+1}, z^{k+1}).$$

Note that $\text{single}_{k}$ is defined as the injection of $a^k \in D_k$ into the singleton $\{a^k\} \in D_{k+1}$:

$$\text{single}_{k}(a^k, a^{k+1}) \equiv a^k \in a^{k+1} \land \forall x^k. x^k \in a^{k+1} \rightarrow x^k = a^k.$$

Assume that the binary input to the Turing machine has length $n$, and define the set $X \subseteq \{0, 1, \ldots, n-1\}$ to be the bit positions of the input that are set to 1. Also, assume that the Turing machine head never moves right of its initial position, and that 0 is the initial state. Let formula $\text{tape}_{i}$ be $c_i \in u^{k+1}$ if $i \in X$, and $c_i \notin u^{k+1}$ otherwise. The relation specifying the initial configuration can then be defined as

$$\text{INITIAL}(z^{k+1}) \equiv \exists u^{k+1}.\exists h^k.\exists s^k.\exists t^k.
\text{ID}(u^{k+1}, h^k, s^k, t^k, z^{k+1}) \land \text{zero}_{k}(h^k) \land \text{zero}_{k}(s^k) \land \text{zero}_{k}(t^k) \land
\exists c_0^k.\text{zero}_{k}(c_0^k) \land \text{tape}_{0} \land
\exists c_1^k.\text{succ}_{k}(c_0^k, c_1^k) \land \text{tape}_{1} \land \exists c_2^k.\text{succ}_{k}(c_1^k, c_2^k) \land \text{tape}_{2} \land \cdots
\exists c_{n-1}^k.\text{succ}_{k}(c_{n-2}^k, c_{n-1}^k) \land \text{tape}_{n-1} \land
\forall b^k. c_{n-1}^k < b^k \rightarrow b^k \notin u^{k+1}.$$

Observe that constants $c_{2i}^k$ could be some fixed variable $x^k$, and the $c_{2i+1}^k$ could be $y^k$. This coding trick reduces the specification of the initial configuration to length $O(n)$, where a binding $\exists c_0^k.\exists c_1^k.\cdots.\exists c_{n-1}^k\cdots$ would make the formula grow as $\Omega(n \log n)$.

Since the Turing machine simulation runs for $K_j(n)$ steps, we need to be able to specify that number via a succinct formula: we define the time bound as a predicate $T_j(t^{j+2})$, meaning that $u^{j+2}$ codes $K_j(n)$:

$$T_{0}(t^2) \equiv \text{con}_{k}^k(t^2)
T_{j}(t^{j+2}) \equiv \forall b^{j+1}. b^{j+1} \in t^{j+2} \leftrightarrow T_{j-1}(b^{j+1}).$$

If $\text{HALT}$ is the constant naming the halting state, the relation defining an accepting configuration is then

$$\text{FINAL}(z^{k+1}) \equiv \exists u^{k+1}.\exists h^k.\exists s^k.\exists t^k.\text{ID}(u^{k+1}, h^k, s^k, t^k, z^{k+1}) \land \text{con}_{k}^{\text{HALT}}(s^k) \land T_{e}(t^k).$$
Moreover, \( z_{k+1} \) and \( \text{final}(z_{k+1}) \) have length \( \Theta(n) \) because they code the length of the input, and the time bound of the computation, each of which is a function of \( n \). (The latter could be reduced at the expense of more complicated coding.) All the other components of the specification have length \( O(1) \).

The definition of the transition relation \( \delta \) is equally straightforward:

\[
\delta(z_{k+1}, z_{k+2}) \equiv \exists u_{k+1}^1, \exists h_1^k, \exists s_1^k, \exists t_1.
\]

\[
\exists u_{k+1}^2, \exists h_2^k, \exists r_2^k.
\]

\[
\text{id}(u_{k+1}^1, h_1^k, s_1^k, t_1, z_{k+1}) \land \text{id}(u_{k+1}^2, h_2^k, r_2^k, z_{k+1}^2) \land
\]

\[
\text{succ}(t_1, t_2) \land \Phi_{\text{TM}}(u_{k+1}^1, t_1, s_1^k, u_{k+1}^{k+1}, h_2^k, r_2^k).
\]

The variables \( z_{k+1} \) and \( z_{k+2} \) code successive IDs of the Turing machine; \( \Phi_{\text{TM}} \) codes their relationship as defined by the transition rules of the machine.

It remains only to define the reflexive, transitive closure of \( \delta \):

\[
\delta^*(u^{k+1}, v^{k+1}) \equiv \exists c^{k+2}.
\]

\[
\forall (r^{k+1}, s^{k+1}).
\]

\[
[(r^{k+1}, s^{k+1}) \in c^{k+2} \leftrightarrow
\]

\[
r^{k+1} = s^{k+1} \lor \exists q^{k+1}, (r^{k+1}, q^{k+1}) \in c^{k+1} \land \delta(q^{k+1}, s^{k+1})]
\]

\[
\land (u^{k+1}, s^{k+1}) \in c^{k+2}.
\]

The constant \( c^{k+2} \) codes the set of pairs \( (r^{k+1}, s^{k+1}) \) that form the transitive closure of the relation defined by \( \delta \).

5. MAIN RESULTS

We summarize the salient features of our above coding in the following theorem.

**Theorem 5.1.** Let \( M \) be a fixed Turing machine that accepts or rejects an input \( x \) in \( k_\ell(|x|) \) steps. Then there exists a formula \( \Phi_x \) in higher-order logic such that \( M \) accepts \( x \) if and only if \( \Phi_x \) is true. Moreover, \( \Phi_x \) only quantifies over universes \( D_\ell \) for \( i \leq \ell + 4 \), and \( |\Phi_x| = O(|x|) \).

By considering the coding of this formula in the simply typed \( \lambda \)-calculus, using the translation described in Sections 4.3 and 4.4, we derive the following corollary:

**Corollary 5.2.** Let \( M \) be a fixed Turing machine that accepts or rejects an input \( x \) of length \( n \) in \( K_\ell(n) \) steps. Then there exists an explicitly typed, closed \( \lambda \)-term \( \Phi_x : \text{Bool} \) such that \( M \) accepts \( x \) if and only if \( \Phi_x \) reduces to true \( \equiv \lambda t : o. \lambda f : o.t : \text{Bool} \). Moreover, the bound variables in \( \Phi_x \) have order \( 2\ell + 13 \), and \( |\Phi_x| = O(n) \).

**Proof.** From Theorems 4.1 and 5.1, we know the largest \( D_\ell \) we need is \( D_{\ell+4} \), coded as \( \lambda \)-term \( D_{\ell+4} \) with other \( 2\ell + 13 \) length \( O((b + 2\ell + 9)!)) \). But recall \( \ell \) is a constant and \( b = \lceil \frac{\log n}{\log \log n} \rceil \); a straightforward calculation shows that for any fixed constant \( c \), \( (b + c)! \leq 2^{b+c} n \); observe that \( (b + c)! \leq 2^{(b+c)!} \leq 2^{(b+c)!} \) and

\[
(b + c) \log(b + c) = (b + c) \left( \log b + \log \left( 1 + \frac{c}{b} \right) \right) \leq b \log b + c \log b + c + o(1),
\]

where

\[
b \log b + c \log b = \log n - \frac{\log n \log \log n}{\log \log n} + c \log n - c \log \log n \leq \log n
\]

since the second term dominates the third term for large \( n \).

Finally, by abstracting over the constructions in Theorem 5.1, the length of the overall term can also be bounded as \( O(n) \).
Combining the last two results, we derive our most important theorem:

**Theorem 5.3 (Main Theorem).** Let \( \ell \geq 0 \) be any fixed integer. Then there exists a set of explicitly typed, closed \( \lambda \)-terms \( E_n : \text{Bool} \), where \( |E_n| = O(n) \), \( E_n \) normalizes in \( O(|E_n|) \) parallel \( \beta \)-steps, and the time needed to implement the parallel \( \beta \)-steps, on any first-class machine model \([\text{vEB}90]\), grows as \( \Omega(K_{\ell}(n)) \).

**Proof.** Let \( \hat{\Phi} \), be a typed, closed \( \lambda \)-term that reduces to \( \text{true} \equiv \lambda t : o. \lambda f : o. t : \text{Bool} \) if and only if an arbitrary, fixed Turing machine \( M \) accepts \( x \) in \( K_{\ell+1}(|x|) \) steps, where the coding of \( \hat{\Phi} \) is given by Corollary 5.2. By that Corollary and the previous Theorem 5.1, we know that the length of \( \hat{\Phi} \) is \( O(|x|) \).

By Theorems 3.12 and 3.13, the number of parallel \( \beta \)-steps needed to normalize \( \hat{\Phi} \) is also \( O(|x|) \).

Suppose now that these parallel \( \beta \)-steps could be implemented by an algorithm in \( K_{\ell}(|E_n|) \) steps. We then would derive the following contradiction: for some constant \( k \geq 0 \),

\[
\text{dtime}[K_{\ell+1}(|x|)] \subseteq \text{dtime}[K_{\ell}(k|x|)].
\]

The bound on the succinctness of \( |E_n| \) ensures that this containment is a contradiction. ■

Certainly the Main Theorem has to apply to the “machine” defined by Lamping’s algorithm, since the graph reduction operations can be easily implemented with a complexity that is polynomial in the number of such operations; hence the graph reduction “machine” is first-class. We derive, as a consequence, the following bound on the number of graph reduction rules required to implement optimal reduction:

**Corollary 5.4.** Let \( \ell \geq 0 \) be any fixed integer. Then there exists a set of \( \lambda \)-terms \( E_n : \text{Bool} \) which normalize in \( O(n) \) parallel \( \beta \)-steps, where the number of local graph interaction steps required to normalize \( E_n \), using Lamping’s graph reduction algorithm, grows as \( \Omega(K_{\ell}(n)) \).

In the previous corollary, the number of graph reduction steps counts not only interactions between \( \lambda \), apply, and sharing nodes, but also interactions involving the croissant and bracket nodes used to manage the indices that control the behavior of the sharing nodes. For more information on how this index management works, see \([\text{LM}96, \text{AG}98]\).

Finally, in renditions of optimal graph reduction rules, there is some ambiguity defining where reduction ends, and where readback begins. For example, if a graph has no more \( \beta \)-redexes, and thus represents a normalized \( \lambda \)-term, one way to read back the term is to continue graph reduction “nonoptimally,” allowing forbidden duplication of \( \lambda \)- and apply-nodes. We may then ask: given a graph without \( \beta \)-redexes, how hard is it to read back the normal form?

Recall the question “Does Turing machine \( M \) accept \( x \) in \( K_{\ell+1}(|x|) \) steps?” The answer to this question can be coded in a sharing graph that can be produced in time polynomial in \( |x| \) (we regard \( M \) and \( \ell \) as fixed), where the structure of this graph is given by the “blob” in Fig. 1. As a consequence, we derive the following corollary:

**Corollary 5.5 (Readback Lemma).** Let \( \ell \geq 0 \) be any fixed integer. Then there exists a set of sharing graphs \( G_n \), where \( \|G_n\| \) is bounded by a small fixed polynomial in \( n \), \( G_n \) contains no \( \beta \)-redexes, the \( \lambda \)-term coded by \( G_n \) has constant size, and the computational work required to read back the represented term grows as \( \Omega(K_{\ell}(n)) \).

All of the above results depend on the reduction of \( \lambda \)-terms, where the number of ordinary \( \beta \)-steps, dwarfs the number of parallel \( \beta \)-steps. We summarize this difference in this final corollary:

**Corollary 5.6.** Let \( \ell \geq 0 \) be any fixed integer. For any normalizing \( \lambda \)-term \( E \), define \( b_E \) to be the number of ordinary \( \beta \)-steps taken and \( p_E \) to be the number of parallel \( \beta \)-steps taken in a reduction of \( E \) to normal form. Then there exists an infinite set of \( \lambda \)-terms \( E \) where \( b_E = \Omega(K_{\ell}(p_E)) \).

**Proof.** Let \( \lambda \)-term \( E \) be the \( \eta \)-expanded version of \( \bar{n}^{2} \cdots 2 \), where \( \bar{m} \) is the Church numeral for integer \( m \), and \( \bar{n} \) is followed by \( \ell + 3 \) copies of \( 2 \). Because \( |E| = O(n) \) we know that \( p_E = O(n) \). On the other hand, \( E \) normalizes to \( K_{\ell+3}(n) \), and an ordinary \( \beta \)-step can at most square the length of a \( \lambda \)-term, hence \( n^{2^{\ell+1}} = \Omega(K_{\ell+3}(n)) \), from which we derive \( b_E = \Omega(K_{\ell+1}(n)) \); hence \( b_E = \Omega(K_{\ell}(p_E)) \). ■
The corollary answers a question raised by Frandsen and Sturtivant in [FS91], who exhibited a set of λ-terms, where \( b \geq 5p \), and conjectured that a set of λ-terms existed where \( b = \Omega(2^p) \). In [LM96] a set of terms was constructed where \( b = \Omega(2^{2^p}) \).

We remark also that the Main Theorem gives bounds on the complexity of cut elimination in multiplicative-exponential linear logic (MELL), and in particular, an understanding of the “linear logic without boxes” formalism in [GAL92b]. In proof nets for linear logic (see, for example, [Laf95, AG98]), the \textit{times} and \textit{par} connectives of linear logic play essentially the same role as \textit{apply-} and \( \lambda \)-nodes in \( \lambda \)-calculus; the programming synchronization implemented by the \textit{closure} has its counterpart in proof net \textit{boxes}. Just as Lamping’s technology can be used to optimally duplicate closures, it can be used to optimally share boxes. Because the simply typed \( \lambda \)-calculus is happily embedded in multiplicative-exponential linear logic, we get similar complexity results: small proof nets with polynomially bounded number of cuts, but a nonelementary number of “structural” steps to resolve proof information coded by sharing nodes.

6. SUPERPOSITION AND HIGHER-ORDER SHARING

The analysis of the previous section makes it very clear that the parallel \( \beta \)-step is not even remotely a unit-cost operation. The workhorse of any optimal reduction engine is not the parallel \( \beta \)-step, but the bookkeeping overhead of sharing redexes. The bookkeeping must remember which redexes are being so parsimoniously shared, transmitting relevant context information between the redexes and the \textit{sanctum sanctorum} where parallel \( \beta \)-reduction takes place.

We used Lamping’s technology as a calculating device to prove a theorem about all possible implementations of optimal evaluation. However, it is worth asking the following question: what are Lamping’s graphs doing to so cleverly encode so much sharing? We ought to get some naive, straightforward, algorithmic understanding of how his data structures are working. In this section, we discuss two essential phenomena of these data structures: that of \textit{superposition} and that of \textit{higher-order sharing}. Superposition is a coding trick where graphs for different values—for example, \textit{true} and \textit{false}, or different Church numerals—are represented by a single graph. Higher-order sharing is a device where sharing nodes can be used to code other sharing structures, allowing a combinatorial explosion in the size of graphs. Both of these techniques are used to realize the generic simulations of the previous section.

It is worth noting that these coding tricks were discovered after the proof of the nonelementary lower bound. It was the proof of that theorem that made us realize such codings had to exist, as opposed to inventing graph gadgets and then trying to embed them in \( \lambda \)-terms. We have resisted stating theorems in this section, instead encouraging the reader toward a more intuitive grasp of the finitary dynamics of the graph reduction rules.

6.1. Superposition

In Fig. 17, we see a really simple, but canonical example of superimposed graphs: the coding of \textit{true} and \textit{false}. Notice how the “star” sides of the sharing nodes code \textit{true}, and the “circle” sides of

![FIG. 17. Superposition of true and false.](image-url)
the sharing nodes code \textit{false}. The two values share the \lambda-nodes that serve as the interface to the external context. A curious consequence of this superposition is that, for example, we can negate both Boolean values at the cost usually ascribed to negating only \textit{one} of them: consider reducing a graph where the function associated with an apply-node is \texttt{not}, and the associated argument is the superimposed \texttt{true} and \texttt{false}: the \texttt{not} function merely switches the topological position of two of the sharing nodes.

While this coding looks compelling, it is worth asking: could such a sharing graph really occur in reducing a term? Consider the coding of $D_1 \equiv \lambda c : \texttt{Bool} \to o \to o.\lambda n : o.\texttt{true} (c \texttt{false} n); (\texttt{Bool} \to o \to o) \to o \to o$, and its $\eta$-expanded equivalent, pictured in Fig. 18. In the unexpanded term, there is a sharing of $c$, but not a sharing of the four applications. In the $\eta$-expanded term, the coding of $c \texttt{true}$ and $c \texttt{false}$ is shared by a single application, and the argument is exactly our superimposed \texttt{true} and \texttt{false}. In addition, there is a sharing of $(c \texttt{true}) (c \texttt{false} n)$ and $(c \texttt{false}) n$. Notice that the companion argument to the superimposed values is just two $\eta$-expansions: the “star” one, coding $(c \texttt{true}) (c \texttt{false} n)$, leads back into the network of applications (but on the “circle” side), while the “circle” application leads out to the parameter $n$.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig18.png}
\caption{\textit{\eta}-Expansion of the list of Booleans.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig19.png}
\caption{Sharing graph representations of Church numerals.}
\end{figure}
This example explains why \texttt{SATISFIABILITY} can be coded in a subexponential number of parallel $\beta$-steps. Consider the formula $\exists x. \exists y. \exists z. \phi(x, y, z)$, represented by the typed $\lambda$-term

\[
D_1(\lambda x : \text{Bool} \cdot \text{OR} \cdot D_1(\lambda y : \text{Bool} \cdot \text{OR} \cdot D_1(\lambda z : \text{ Bool} \cdot (\phi \ x \ y \ z) \ false) \ false) \ false).
\]

Generalizing the above graph constructions, this kind of $\lambda$-term essentially superimposes every row of an $n$-variable truth table into a single, shared structure, so that applying a Boolean function to all $2^n$ rows can be done “simultaneously”.

Another beautiful example comes from Church numerals. In Fig. 19, we see two sharing graphs, representing the Church numerals for 2 and 3, in which every application of the “successor” constructor $s$ is shared. Notice that these examples give us a fairly universal picture of Church numerals, where the $\lambda s$ and $\lambda z$ serve as a uniform interface, the applications are maximally shared, and two numerals are distinguished only by their network of sharing nodes: see Fig. 20. Now we can code a superimposed representation of many Church numerals, by inserting more sharing nodes to serve as multiplexors and demultiplexors leading to the correct sharing network pictured in Fig. 21.
This style of computation looks like a $\lambda$-calculus optimal evaluation implementation of SIMD (single instruction, multiple data) parallelism. Imagine a list $\lambda c.\lambda n.cx_1 (c x_2 \cdots (c x_n)\cdots)$, where the $x_i$ are Church numerals, which gets $\eta$-expanded so that the applications $(c x_i)$ are all shared, via superimposed representations of the numerals. We can then, for example, apply any integer function to each of the numerals “simultaneously,” that is, counting parallel $\beta$-steps, but not counting sharing interactions. The sharing network for each numeral is like a separate processor, serving as multiple data; the $\lambda$- and apply-nodes serve as inter-faces to the external context; the multiplexers and demultiplexers replicate and pump the single-instruction stream (the code for, say, factorial) to each of the processors. The real work of the computation becomes communication, performing the $\eta$-expansion on the lists, and communicating the function to different processors. But the actual computation associated with the function occurs via interaction of sharing nodes only, our “parallel computation.”

6.2. Higher-Order Sharing

Superposition is only one component of the graph reduction technology that supports nonelementary computation in a trivial number of parallel $\beta$-steps. The other essential phenomenon is higher-order sharing, used to construct enormous networks of sharing nodes.

A well-understood use of sharing appears in [Asp96, LM96], where a linear number of sharing nodes is used to simulate an exponential number of function applications, illustrated in Fig. 22.

The key idea in this construction, and in more sophisticated examples of higher-order sharing, is the pairing of sharing nodes with wires from the “circle” to “star” sides, so that we can enter the same graph in two different “modes.” In fact, a trivial example of the “geometry of interaction” as interpreted in sharing graphs occurs via a simple stack semantics, where a naive version of context information can be constructed as mere stacks of circle and star tokens. When a stack enters a sharing node at its interaction port, the top token on the stack is popped and used to determine whether the path to follow is along the “circle” or “star” auxiliary ports of the node. Conversely, when entering a sharing node via its circle (respectively, star) auxiliary port, a circle (respectively, star) token is pushed on the stack. The semantics of sharing graphs, essentially represented by this stack discipline, is preserved by graph reduction.

FIG. 22. Binary counting and exponential function application.

FIG. 23. A simple sharing network.
In the left part of Fig. 22, we see a network made of $2k$ sharing nodes that implements a path of length $2^k$ using the naive stack semantics. If the stack of circle and star tokens is thought of as a binary number, then the path corresponds to binary counting. In the second part of the figure, we use the same construction to implement $2^k$ applications of a function $f$ to an argument $x$.

However, this construction merely shares an application node; to get a truly powerful network of nodes, we need to be able to share sharing nodes as well. The construction is indicative; the basis is described by the simple network as in Fig. 23, where two binary trees annihilate. By linking two symmetric copies of this construction via the marked wire, we get a “stack” of sharing nodes implementing the standard exponential construction. This structure, used in the previous figure to share a function application, can instead be used to share a sharing node, as illustrated in Fig. 24.

Now we can go one step further, considering a nested construction, where we share the sharing of sharing nodes, ad nauseum. Figure 25 hints at how such a network of $2^\ell$ sharing nodes can expand to a network of size $K_\ell(1)$.

After these illustrative examples, we can describe the basic idea of the elementary construction (in the sense of Kalmár). We define a sequence of networks $N_j$, where $N_j$ contains sharing nodes at levels (not to be confused with indices) $0 \leq \ell \leq j$. Like a sharing node, every network will have one principal port, two auxiliary ports (identified as the circle and star ports), and a distinguished sharing node that we will call the core node. Given a sharing network $N$, we will write $\bar{N}$ to mean $N$ with circle and star exchanged on the auxiliary ports of the core node of $N$.

The network $N_0$ is a single sharing node at level 0, which is by process of elimination the core node. To construct $N_{j+1}$, we combine $N_j$, $\bar{N}_j$, and a new core node at level $j + 1$, attaching the principal...
ports of the core node and $\overline{N}_j$, the principal port of $N_j$ to the circle auxiliary port of the core node, and the star auxiliary node of $N_j$ to the circle auxiliary node of $\overline{N}_j$ (see Fig. 26) The principal port of $N_{j+1}$ is the star external port of $\overline{N}_j$; auxiliary ports are the star auxiliary port of the core node, and the circle auxiliary port of $N_j$. It should be clear that $N_j$ has $2^j - 1$ sharing nodes. We define a naive oracle for interactions between sharing nodes: nodes at identical level annihilate; otherwise they duplicate.

**Lemma 6.1.** The network $N_j$ (respectively, $\overline{N}_j$) normalizes to a complete binary tree with $K_j(1)$ leaves. The leaves at level $j$ are connected to sharing nodes at level $j - 1$ on their star (respectively, circle) auxiliary ports; the remaining auxiliary port is connected to the primary port of the node at the adjacent leaf, as in Fig. 26.

**Proof.** By induction. The lemma is trivially true for $j = 0$. For $j = i + 1$, use the induction hypothesis to normalize $N_i$ and $\overline{N}_i$, producing the network in Fig. 26. The two binary trees now annihilate each other, and the two stacks of $K_i(1)$ sharing nodes then create a complete binary tree with $K_{i+1}(1)$ copies of the core node, with a complete binary tree linked to these copies at the leaves. 

This kind of sharing network results from the parallel $\beta$-reduction of the $\eta$-expanded term $2_j = \overline{2} \cdots \overline{2}$ of $j \overline{2}$, where $\overline{2}$ is the Church numeral for 2. This term has length $O(2^j)$ because of explicit type information that doubles for each additional $\overline{2}$; $2_j$ normalizes in $O(j)$ parallel $\beta$-steps to the Church numeral for $K_j(1)$. The exponential length is sufficient to code a copy of $N_j$ and $\overline{N}_j$; after normalization, these networks expand to construct $K_j(1)$ function applications. The same computational phenomenon is evident in the coding of the iterated powerset, though not as straightforward to describe.

7. CONCLUSIONS

We have shown in this paper that the cost of implementing a sequence of $n$ parallel $\beta$-reduction steps in the reduction of a $\lambda$-term is not bounded in general by any Kalmár-elementary function $K_i(n)$. Given that the parallel $\beta$-step is one of the key ideas in optimal reduction, it makes sense to consider whether the idea of optimal evaluation has any practical importance. Do we need a reevaluation of the idea of optimality? There is no question that the study of optimal reduction has provided important insights and connections among linear logic, control structures, context semantics and the geometry of interaction, and intensional semantics, with hope of our applying its ideas as well in the area of full abstraction. But all that taken for granted, it is irrelevant in the world of pragmatics. Is Lamping’s graph reduction algorithm, or any of its variants, any good?

Lower bounds are fertile territory for pessimists whose only happiness in life is to show that the world we typically think of as computable is actually intractible. But suppose we had found instead that the parallel $\beta$-step was indeed unit-cost, or that $n$ such reductions could be carried out in time polynomial
in $n$. In the face of generic simulation, this would mean that there must be \textit{lots} of such reduction steps. The critic would then infer: why all this fancy new inscrutable technology to implement something that is not much better than an ordinary $\beta$-step? Needless to say, there is much ground in between these two extremes where similar pessimism could be expressed.

As we wrote in the Introduction, the computation theorist’s idea of \textit{generic simulation} is comparable with the classical physicist’s idea of \textit{work}; just as real work requires an expenditure of energy, generic simulation requires an unavoidable commitment of computational resources. What we do learn, beyond the shadow of a doubt, is that merely minimizing the number of parallel $\beta$-steps is a curiosity at best, and a worst a misleading diversion from the business of designing efficient interpreters for programming languages. If there is something efficient about Lamping’s graph reduction algorithm, it must be something else.

Along these lines, we do believe that the graph reduction algorithm has something good to say for itself: it maximizes sharing. Furthermore, it has the potential to optimize bookkeeping (croissant and bracket interactions) in its incremental manipulation of the boxes which delineate sharable values. Optimality considerations require that similar $\beta$-redexes are shared, but many other subexpressions are also shared. In particular, subexpressions with the same Lévy label are almost always shared.

This insight and intuition can be formalized more precisely in the following sense. In the labeled $\lambda$-calculus, the set of labels is made up of a countably infinite set of atomic labels (such as integers) and closed under concatenation and underlining. Each subterm of the initial $\lambda$-term is initially annotated with a unique label; as reduction occurs, labels are concatenated and underlined, according to the arcane rule

$$(\lambda. \lambda. E) \rightarrow E[x \mapsto F].$$

For example, in the untyped $\lambda$-calculus, we have the labeled reduction step

$$((\lambda. (x^1 \cdot x^2 \cdot x^3)^4 \cdot (\lambda. (x^5 \cdot x^6)^7)^8)^9 \rightarrow ((\lambda. (x^5 \cdot x^6)^7)^81 \cdot (\lambda. (x^5 \cdot x^6)^7)^82)^89).$$

A labeled $\lambda$-term thus codes the reduction history that led to its derivation.

It can be shown that the number of interactions of $\lambda$-nodes, apply-nodes, and sharing nodes in optimal graph reduction is bounded by a polynomial in the number of \textit{unique} labels generated in a reduction [LM97]. If we believe that label identifies a set of similar subexpressions, this result means that the graph reduction algorithm is maximizing sharing up to a polynomial factor. This is probably a good thing. It remains, however, to solve the unquestioned algorithmic problem of \textit{spurious bracketing}—the explosion of so-called \textit{bracket} and \textit{croissant} nodes that implement \textit{node indices} and the so-called “oracle” that implements interactions between sharing nodes.

Many questions remain about the complexity of this graph technology. A complementary tight upper bound on the number of sharing interactions is not known. It is still not known whether the cost of spurious \textit{bracket} and \textit{croissant} interactions can be greatly reduced, or whether they are essential, even with the optimization possible from so-called “safe nodes.” We conjecture that, with proper optimization, the bookkeeping is polynomial in the number of fan interactions. It also remains to be seen whether the $\eta$-expansion method could be generalized to higher-order typed $\lambda$-calculi, particularly System F [GLT89], which might give new and constructive proofs of strong normalization for those calculi.

In summary, graph reduction for optimal evaluation presents a new technology for language implementation that is a hybrid of call-by-name and call-by-value. Its theoretical importance is unquestioned, and its practical impact has only just begun to be considered. When Zhou En-Lai was asked to comment on the importance of the French Revolution, he reportedly responded, “It’s too early to tell.” In considering the importance of sharing graphs as a new language implementation technology, we are encouraged to say the same thing.

ACKNOWLEDGMENTS

For many relevant and fruitful discussions, suggestions, corrections, admonishments, and encouragements, we thank Arvind, Stefano Guerrini, Paris Kanellakis, Jakov Kucan, Julia Lawall, Jean-Jacques Lévy, Simone Martini, Albert Meyer, Luke Ong, Luca Roversi, and René Vestergaard.
REFERENCES


