Parallel beta reduction is not elementary recursive

*Andrea Asperti**
Dipartimento di Scienze dell'Informazione
Via di Mura Anteo Zamboni
40127 Bologna, Italy
asperti@cs.unibo.it

Harry G. Mairson†
Computer Science Department
Brandeis University
Waltham, Massachusetts 02254
mairson@cs.brandeis.edu

Abstract

We analyze the inherent complexity of implementing Lévy's notion of *optimal evaluation* for the λ-calculus, where similar redexes are contracted in one step via so-called *parallel β-reduction*. Optimal evaluation was finally realized by Lamping, who 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.

We prove that the cost of parallel β-reduction is not bounded by any Kalmár-elementary recursive function. Not merely do we establish that the parallel β-step cannot be a unit-cost operation, we demonstrate that the time complexity of implementing a sequence of n parallel β-steps is not bounded as $O(2^n)$, $O(2^{2^n})$, $O(2^{2^{2^n}})$, or in general, $O(\mathrm{K}_\ell(n))$ where $\mathrm{K}_\ell(n)$ is a fixed stack of ℓ 2s with an n on top.

A key insight, essential to the establishment of this nonelementary lower bound, is that any simply-typed λ-term can be reduced to normal form in a number of parallel β-steps that is only polynomial in the length of the explicitly-typed term. The result follows from Statman's theorem that deciding equivalence of typed λ-terms is not elementary recursive.

The main theorem gives a lower bound on the work that must be done by *any* technology that implements Lévy's notion of optimal reduction. However, in the significant case of Lamping's solution, we make some important remarks addressing *how* work done by β-reduction is translated into equivalent work carried out by his bookkeeping nodes. In particular, we identify the computational paradigms of *superposition* of values and of *higher-order sharing*, appealing to compelling analogies with quantum mechanics and SIMD-parallelism.

*Partially supported by Esprit WG-21836 CONFER-2.
†Supported by ONR Grant N00014-93-1-1015, NSF Grant CCR-9619638, and the Tyson Foundation.

POPL 98 San Diego CA USA

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 implementors 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 *labelled* 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 *shared*, and evaluated *at once* (via a so-called *parallel β-reduction*) by any efficient scheme.

Recent research by Lamping, and independently Kathail, has shown that there indeed exist λ-calculus evaluators satisfying Lévy's specification [Lam90, Kat90]. Lamping introduced a beautiful graph reduc-

tion 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, Abadi, and Lévy [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 form by others, is to understand the computational complexity of sharing as a function of the "real work" of β-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 β-reduction by *any* implementation technology. In this paper, we take major steps towards resolving such global questions, with important algorithmic insights into the relevant graph reduction technology. Specifically, we prove that the cost of parallel β-reduction is not bounded by any Kalmár-elementary recursive function. Not merely do we establish that a parallel β-step is not a unit-cost operation, we prove that the time complexity of implementing a sequence of n parallel β-steps is not bounded as $O(2^n)$, $O(2^{2^n})$, $O(2^{2^{2^n}})$, or in general, $O(\mathrm{K}_\ell(n))$ where $\mathrm{K}_\ell(n)$ is a fixed stack of ℓ 2s with an n on top.

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

Definition 1.1 *Let E be a labelled λ-term that normalizes in n parallel β-steps. Let algorithm* A *be an interpreter that normalizes the unlabelled equivalent of λ-term E in t time steps. We then say that* A implemented the n parallel β-steps at cost t, *and define the* cost of a single parallel β-step *in this reduction as 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 β-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 β-step must require cost t/n.

A key insight, essential to the establishment of our nonelementary lower bound, is that any simply-typed λ-term can be reduced to normal form in a number of parallel β-steps that is only linear in the length of the explicitly-typed term. The proof of this claim depends on the judicious use of *η-expansion* to control the number of parallel β-steps. Not only does η-expansion act as an *accounting mechanism* that allows us to see order in the graph reduction, it moreover serves as a lovely sort of *optimizer* that exchanges the work of parallel β-reduction for the work of *sharing*. Our result then follows from Statman's theorem that deciding equivalence of typed λ-terms is not elementary recursive [Sta79]. We emphasize in Statman's theorem the *generic simulation* of time-bounded computation, proved using the 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.

Definition 1.2 *We define the* Kalmár-elementary functions $\mathrm{K}_\ell(n)$ *as* $\mathrm{K}_0(n) = n$, *and* $\mathrm{K}_{\ell+1}(n) = 2^{\mathrm{K}_\ell(n)}$ *[Kal43]. Dually, we define the* iterated logarithm functions $\log^{(m)}(n)$ *as* $\log^{(0)}(n) = n$, *and* $\log^{(c+1)}(n) = \log(\log^{(c)}(n))$.

Main Theorem. *There exists a set of closed λ-terms E_n : Bool, where $|E_n| = O(n \log^{(c)} n)$ for any integer $c > 0$, such that E_n normalizes in $O(|E_n|)$ parallel β-steps, and the time needed to implement the parallel β-steps,* on any first-class machine model[1], *grows as $\Omega(\mathrm{K}_\ell(n))$ for any fixed integer $\ell \geq 0$.*

Proof Sketch. For a fixed Turing Machine M, and input x of length n, consider the question, "Does M accept x in $\mathrm{K}_{\ell+1}(n)$ steps?" We show how to compile this question into a *succinct* simply-typed λ-term E, where the length of the explicitly-typed term E is exponential in ℓ and polynomial in n. Since $\ell+1$ is the fixed height of the "stack of 2s," the exponential factor makes no asymptotic difference: it is only a constant. The term E reduces to the standard λ-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 β-steps. Suppose that the $|E|$ parallel β-steps could indeed be implemented at time cost $\mathrm{K}_\ell(|E|)$; we would then have shown that

$$\textsc{dtime}[\mathrm{K}_{\ell+1}(|x|)] \subseteq \textsc{dtime}[\mathrm{K}_\ell(|E|)].$$

But the time hierarchy theorem from complexity theory (see, e.g., [HU79]) tells us that this implied conclusion is

[1] 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.

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

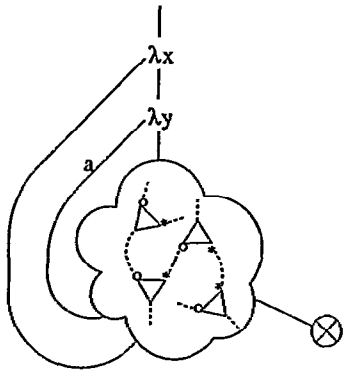


Figure 1: The blob.

on the number of parallel β-steps is proven by computing the reduction of E using Lamping's algorithm. We use Lamping's technology as a *calculating 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 inscrutable, in the labelled λ-calculus.

In particular, when the λ-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 Figure 1. It looks just like the graphs for the λ-calculus "true" or "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 Figure 1 connect (respectively) to the λx and λy 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-β 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 ten 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 ten thousand equations: Feynman was able to diagramatically work out calculations that seemed interminable by more formal means (see, e.g., [Dys79, Sch94]). Lamping's graphs give insight into the inherent process of optimal reduction that transcends the particularities of his implementation technology.

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 analogue of simply-typed lambda calculus.

In the significant case of Lamping's solution, we can also make some important remarks addressing *how* work done by β-reduction is translated into equivalent work carried out by his bookkeeping nodes. We identify the computational paradigms of *superposition of values* and of *higher-order sharing*, appealing to compelling analogies with quantum mechanics and SIMD-parallelism.[2]

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 β-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 β-steps are *necessarily* resource-intensive.

We present in Section 2 an explanation of the graph reduction technology, and in Section 3 a description of the η-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 λ-terms. Section 5 contains the main results of the paper. Finally, Section 6 describes the basic graph constructions involving sharing nodes that, in Lamping's algorithm, allow huge

[2] In choosing the rubric of *superposition*, we recall how Claude 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]

computations to be simulated by so few parallel β-steps.

2 Lamping's graph reduction technique

Lamping's algorithm was designed for the *optimal* handling of *shared redexes* in the evaluation of λ-expressions. 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 $(\lambda f.\lambda z.z(fM)(fN))(\lambda x.F[x])$, which β-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 expressions 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 exiting through a hole. Lamping's graph reduction algorithm consists of a set of local rewrite rules that can be classified naturally in two groups. First, we have rules involving application, abstraction and sharing, which are responsible for implementing β-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. When two sharing nodes interact, they either duplicate or annihilate each other, depending on local information that is effectively computed by the control nodes.

2.1 Initial translation

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

$$\begin{aligned} \mathcal{T} &:= o \mid \mathcal{T} \to \mathcal{T} \\ \mathcal{V} &:= v_1 \mid v_2 \mid v_3 \mid \cdots \\ \mathcal{E} &:= \mathcal{V} \mid \mathcal{E}\mathcal{E} \mid \lambda\mathcal{V} : \mathcal{T}.\mathcal{E} \end{aligned}$$

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 $\alpha \to \beta \to \gamma$ means $\alpha \to (\beta \to \gamma)$, and term EFG means $(EF)G$.

Associated with each *well-typed* λ-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 τ. 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 α as declared in the *binding* $\lambda x : \alpha \ldots$; every subterm $\lambda x : \alpha.B$ has type $\alpha \to \beta$ when β is the type of B; and EF has type β when E has type $\alpha \to \beta$ and F has type α.

In optimal graph reduction, a λ-term is initially represented by a graph similar to its abstract syntax tree; however, we introduce an explicit node for sharing, and an explicit connection between variable occurrences (represented by wires) and the λ-nodes that represent their respective binders. We shall label each wire edge of the graph with a suitable type; for brevity, we shall often use the notation β^α instead of $\alpha \to \beta$. These type annotations have no operational role in reduction of the λ-term; like the names of the planets, they are merely an invariant that provide information to the analyst.

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 a 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 consequence, only a single wire edge, at the root of the binary tree, connects the λ-node representing a binding to the wires representing the variable occurrences.

Each node in the graph (apply, λ and fan) has exactly three distinguished ports where it can be connected to other ports. One of this ports (depicted with an arrow in Figure 2) is called *principal*: it is the only

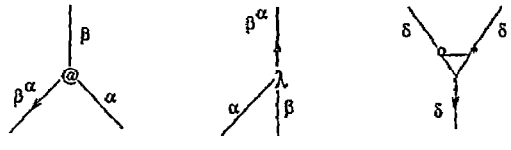


Figure 2: Nodes, ports and their types.

port where the node may interact with other nodes; see the interaction rules in the Appendix. The other ports are called *auxiliary* ports. The sharing graph, and the corresponding term it represents, is 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 Figure 2, whose interpretation should be clear.

3 The η-expansion method

Given a simply typed λ-term E, we show how to construct a variant E' that is $\beta\eta$-equivalent to E, derived

by introducing *η-expansions* of variables occurring 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 β-steps needed to normalize E' is linearly bounded by its size. As a consequence, we demonstrate that the normal form of any simply typed λ-term can be derived in a linear number of parallel β-steps. We write $\|\sigma\|$ for the size of a simple type, counting the size of its tree representation, and $|E|$ for the size of a λ-term, counting 1 for each λ and apply in E, plus $\|\sigma\|$ for each variable of type σ. Finally, we write $[\![G]\!]$ for the number of nodes in a sharing graph G. Variants of these metrics do not change the main theorem we will prove, as long as these alternative metrics are polynomial in the ones we have given.

The bound we prove on the number of parallel reductions depends on controlling the duplication of λ and apply nodes by sharing nodes. When a sharing node has type $\alpha \to \beta$ and faces the principal port of either a λ-node or an apply node, duplication creates two sharing nodes, of types α and β respectively. If the value being shared by a node is the base type o, then that sharing node cannot interact with a λ 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 *η-expansion* is to force a node sharing a value x of type σ to the base type o, by making that node duplicate components of the graph coding the η-expansion of x.

Definition 3.1 *Let x be a variable of type σ. The* η-expansion $\eta_\sigma(x)$ *of x is the typed λ-term inductively defined on σ as: $\eta_o(x) = x$, and $\eta_{\alpha_1 \to \cdots \to \alpha_k \to o}(x) = \lambda y_1 : \alpha_1. \ \ldots \ \lambda y_k : \alpha_k . x \ (\eta_{\alpha_1}(y_1)) \ldots \ (\eta_{\alpha_k}(y_k))$.*

In the graph representation, each η-expanded variable is coded by a subgraph with two distinguished wires called the *positive entry* and the *negative entry* of the variable. When the variable is of base type o, this subgraph is just a wire.

Suppose $\sigma = \alpha_1 \to \ldots \to \alpha_n \to o$. Then the graph G coding $\eta_\sigma(x)$ has the structure depicted in Figure 3, with n λ-nodes and n apply nodes. If a sharing node is placed at the *positive* entry, that node will duplicate the n λ-nodes; copies of the sharing node will move to the auxiliary (non-interaction) port of the top apply node, which has base type o, and to the *negative* entries of the subgraphs G_i representing $\eta_{\alpha_i}(y_i)$. 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 λ-node, which has base type o, and to the *positive* entries of the subgraphs G_i representing $\eta_{\alpha_i}(y_i)$.

The graph representing an η-expanded term has the following nice properties:

Lemma 3.2 *If G is the initial sharing graph representing $\eta_\sigma(x)$, then $[\![G]\!] \leq 2\|\sigma\|$.*

Lemma 3.3 *Let x be a variable of type σ, 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 $[\![\Delta(x)]\!] \leq 3\|\sigma\|$.*

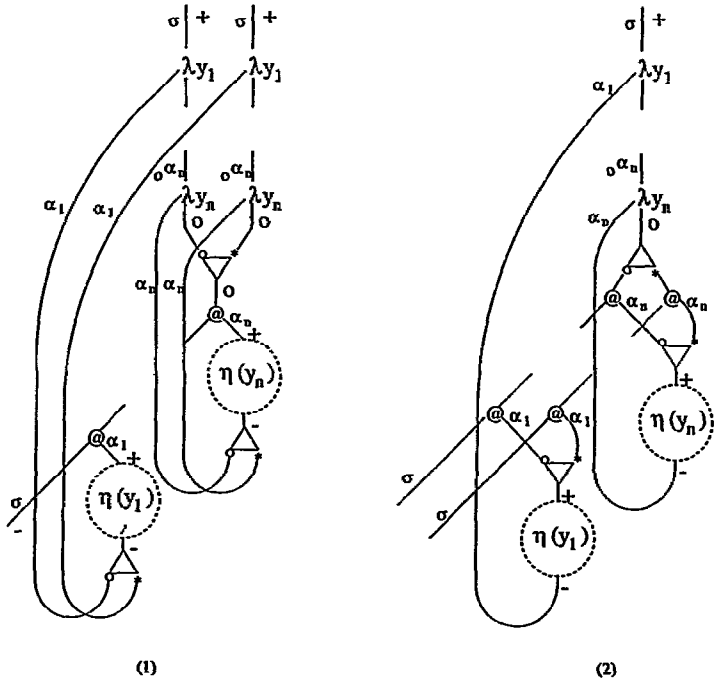


Figure 3: Fan propagation inside $\eta(x)$ with positive and negative entry points.

Lemma 3.4 *Let $\Delta^t(x)$ 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 $[\![\Delta^t(x)]\!] \leq (2+t)\|\sigma\|$.*

We now rewrite the term so as to kill the duplicating power of the sharing nodes, by pushing them immediately through the η-expansion code to base type.

Definition 3.5 *Let M be a simply-typed λ-term. The* optimal root 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 β-redexes introduced by this transformation as* preliminary redexes.

Clearly or$(M) \equiv_{\beta\eta} M$; the transformation duplicates, once and for all, the "skeleton" of this term that describes all of its possible uses. The relevant information about these uses is provided unambiguously by the type σ.

Definition 3.6 *We define $\Delta(M)$ to be the sharing graph obtained from* $\mathrm{or}(M)$ *by reducing all of the preliminary redexes, and propagating all sharing nodes to the base type.*

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

Theorem 3.7 *Let M be a simply-typed λ-term. Then all sharing nodes in $\Delta(M)$ have atomic types, and $[\![\Delta(M)]\!] \leq 2|M|$.*

The reader may be bothered by this "linear" bound, which depends on the definition of the size function $|M|$, where the occurrence of an x of type σ contributes $\|\sigma\|$ to the sum. It depends on how we defined the size of λ-terms; however, its significance—good enough for the lower bound—is that $[\![\Delta(M)]\!]$ *is only polynomial in* $|M|$. As an important consequence of Theorem 3.7, we make the following observations:

Theorem 3.8 *The total number of β-reductions (and thus of families) in the graph normalization of $\Delta(M)$ cannot exceed its initial size.*

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 all logical rules are "below" the structural ones. One can immediately removes the logical cuts, and the rest of the computation is merely structural—the annihilation or duplication of sharing nodes. Theorem 3.8 then yields a familiar theorem:

Theorem 3.9 *The simply typed λ-calculus is strongly normalizing.*

This latter observation is also a straightforward consequence of a result due to Lévy ([Lévy78], Theorem 4.4.6): let Σ be any finite (possibly parallel) reduction of a term M. Then any reduction Σ' *relative*[3] to Σ is terminating.

Since all redexes created along *any* reduction of $\mathrm{or}(M)$ eventually belong to some of its families, any reduction strategy is terminating. The bound on the reduction is, if one counts parallel β-reduction of families, merely linear in the length of the initial term.

We observe finally that the proof of Theorem 3.9 shows a great similarity to an early proof of normalization due to Alan Turing [Gan69]. Define the *index* of a redex $(E^{\sigma\to\tau}F^{\sigma})^{\tau}$ to be $\sigma \to \tau$; Turing's simple observation was that, aside from the copying of existant redexes, the β-reduction of redexes with index $\sigma \to \tau$ can only construct new redexes of index σ or index τ. Normalization then follows by a straightforward induction on types alone. This contrasts with the familiar proof based on the idea of reducibility (see, e.g., [HS87, GLT89, Tai67]).

[3] Σ' is relative to Σ if all redexes in Σ' are in the same family of some redex in Σ, see ([Lévy78], Def. 4.4.1).

4 Simulating generic elementary-time bounded computation

Now that we know that the normal form of a simply-typed λ-term can be computed in a linear number of parallel β-steps, our goal is to construct a *generic reduction* from the largest time hierarchy we can manage via a *logspace reduction*. For example, if we can simulate deterministic computations in $\mathrm{DTIME}[2^n]$ (deterministic exponential time) in the simply-typed λ-calculus, where the initial λ-term corresponding to a computation has length bounded by a fixed polynomial in n, we may then conclude that the parallel β-reduction *cannot* be unit cost. Were a parallel β-step implemented in unit cost, we would have shown that PTIME equals $\mathrm{DTIME}[2^n]$, contradicting the *time hierarchy theorem* (see, e.g., [HU79]). In fact, for any integer $\ell \geq 0$, we can construct generic reductions of this kind for $\mathrm{DTIME}[\mathrm{K}_\ell(n)]$. The consequence is that the cost of a sequence of n parallel β-steps is not bounded by any of the *Kalmár-elementary recursive functions* $\mathrm{K}_\ell(n)$.

Rather than code directly in the simply-typed λ-calculus, 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 λ-calculus [Mey74, Sta79], following the first-principles technology found in [Mai92]. The higher-order logic provides a nice metalanguage to describe typed λ-terms: the logic is extensional, and the λ-term is intensional. We use the logic to describe long computations with short formulas; the compendium of coding tricks reads like the catalogue of primitive recursive functions in Gödel's first incompleteness theorem. Our goal is to define the formula

$$\exists u^{k+3}.\exists v^{k+3}.\mathrm{INITIAL}(u^{k+3}) \wedge \delta^*(u^{k+3}, v^{k+3}) \wedge \mathrm{FINAL}(v^{k+3}),$$

where $\mathrm{INITIAL}(u^{k+3})$ expresses that u^{k+3} codes our desired initial configuration, $\mathrm{FINAL}(v^{k+3})$ expresses that v^{k+3} codes our final, accepting configuration, and $\delta^*(u^{k+3}, v^{k+3})$ 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 $\mathrm{K}_\ell(n)$ on an input to the Turing Machine of size n. Our formula is *succinct*—not much longer than the length of the input to the Turing Machine computation.

To begin, let $\mathcal{D}_1 = \{\mathbf{true}, \mathbf{false}\}$, and define $\mathcal{D}_{t+1} = \mathit{powerset}(\mathcal{D}_t)$. We analyze the complexity of deciding the the truth of formulas with quantification over elements of $\mathcal{D}_t$, for any $t > 0$, using a naive interpretation. Let x^t, y^t, z^t be variables allowed to range over the elements of $\mathcal{D}_t$; we define the *prime formulas* as $\mathbf{true}$, $\mathbf{false}$, x^1, $\mathbf{true} \in y^2$, $\mathbf{false} \in y^2$, and $x^t \in y^{t+1}$. Now consider a formula Φ built up out of prime formulas, the

usual logical connectives $\vee$, $\wedge$, $\rightarrow$, $\neg$, and the quantifiers $\forall$ and $\exists$: is Φ true under the usual interpretation?

Imagine that we code $\mathcal{D}_t$ as a λ-term $\mathbf{D}_t$ which lists all elements of $\mathcal{D}_t$, each coded appropriately as a λ-term of type Δ_t, and we have coded a Boolean function Φ as a λ-term $\hat{\Phi}$ of type $\Delta_t \rightarrow \mathsf{Bool}$. Then, for example, the truth of $\forall x^t.\Phi(x^t)$ can be coded as the λ-term

$$\mathbf{D}_t\ (\lambda x^t : \Delta_t.\textsc{and}\ (\hat{\Phi}\, x^t))\ \mathit{true}.$$

The prime formulas can also be simulated using a similar coding.

4.1 Coding higher-order logic in typed lambda calculus

Let τ be any first-order type, and define $\mathsf{Bool} \equiv o \rightarrow o \rightarrow o$. Define the Boolean values and logical connectives via their usual Church codings. The set $\mathcal{D}_1$ is represented as the list $\mathbf{D}_1$ containing *true* and *false*:

$$\begin{aligned}\mathbf{D}_1 &\equiv \lambda c : \mathsf{Bool} \rightarrow \tau \rightarrow \tau.\lambda n : \tau.c\ \mathit{true}\,(c\ \mathit{false}\, n)\\ &: (\mathsf{Bool} \rightarrow \tau \rightarrow \tau) \rightarrow \tau \rightarrow \tau\end{aligned}$$

We abbreviate the type of $\mathbf{D}_1$ as Δ_1; in general, let $\Delta_{k+1} \equiv \Delta_k^*$, where for any type α, we define $\alpha^* \equiv (\alpha \rightarrow \tau \rightarrow \tau) \rightarrow \tau \rightarrow \tau$, and $\Delta_0 \equiv \mathsf{Bool}$.

Next, for each integer $t > 1$, we define an explicitly-typed λ-term $\mathbf{D}_t$ representing $\mathcal{D}_t$ as a *list* of (recursively defined codings of) all subsets of elements of $\mathcal{D}_{t-1}$ in the type hierarchy. To do so, we must introduce an explicit powerset construction, so as to build succinct terms coding these lists. First, we can define a term *double* where, for example, *double false*[[ ],[*true*]] reduces to [[*false*],[*false*, *true*],[ ],[*true*]]. We may then define

$$\begin{aligned}\mathit{powerset} &\equiv \lambda A^* : (\alpha \rightarrow \alpha^{**} \rightarrow \alpha^{**}) \rightarrow \alpha^{**} \rightarrow \alpha^{**}.\\ &\qquad A^*\ \mathit{double}\\ &\qquad\quad (\lambda c : \alpha^* \rightarrow \tau \rightarrow \tau.\lambda n : \tau.c\\ &\qquad\qquad (\lambda c' : \alpha \rightarrow \tau \rightarrow \tau.\lambda n' : \tau.n')\ n)\\ \mathit{powerset} &: ((\alpha \rightarrow \alpha^{**} \rightarrow \alpha^{**}) \rightarrow \alpha^{**} \rightarrow \alpha^{**}) \rightarrow \alpha^{**}.\end{aligned}$$

The function of *powerset* on lists is like that of *exponentiation* realized via iterated 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: $\mathbf{D}_{t+1} \equiv \mathit{powerset}\ \mathbf{D}_t : \Delta_{t+1} \equiv (\Delta_t \rightarrow \tau \rightarrow \tau) \rightarrow \tau \rightarrow \tau$. We use $\mathbf{D}_t$ as an iterator, where $\mathbf{D}_t$ is typed as $\Delta_t[\tau := \Delta_{t+1}] = (\Delta_{t-1} \rightarrow \Delta_{t+1} \rightarrow \Delta_{t+1}) \rightarrow \Delta_{t+1} \rightarrow \Delta_{t+1}$. The size of $|\mathbf{D}_t|$ can be bounded as follows:

Theorem 4.1 *There exists a fixed constant d such that if $\mathbf{D}_t : \Delta_t$, then $|\mathbf{D}_t| \leq d(2t)!$.*

$|\mathbf{D}_t|$ grows like the factorial of t, unlike the iterated exponential $\overline{2\,2}\cdots\overline{2}$, with length that grows exponentially. In the substitutions $\tau := \Delta_t$ used to type iterated powerset, each substitution multiplies the size of the λ-term by a factor of t, and the cumulative effect is like factorial.

There is a natural idea of *equality* between elements of $\mathcal{D}_k$; 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 $t > 1$, we define terms eq_t, $subset_t$, and $member_t$. When $t = 1$, we define only

$$\begin{aligned}eq_1 \equiv\ & \lambda x^1 : \mathsf{Bool}.\lambda y^1 : \mathsf{Bool}.\\ & \textsc{or}\ (\textsc{and}\ x^1 y^1)\ (\textsc{and}\ (\textsc{not}\ x^1)\,(\textsc{not}\ y^1))\end{aligned}$$

as a basis; eq_1 is just the Boolean 'iff.' For the inductive case of the definitions, let $\Delta_j^{\mathsf{Bool}} = \Delta_j[\tau := \mathsf{Bool}]$, which is the type of an iterator with Boolean output, and define

$$\begin{aligned}member_{t+1} \equiv\ & \lambda x^t : \Delta_t^{\mathsf{Bool}}.\lambda y^{t+1} : \Delta_{t+1}^{\mathsf{Bool}}.\\ & y^{t+1}\ (\lambda y^t : \Delta_t^{\mathsf{Bool}}.\textsc{or}\ (eq_t\ x^t\, y^t))\ \mathit{false}\\ subset_{t+1} \equiv\ & \lambda x^{t+1} : \Delta_{t+1}^{\mathsf{Bool}}.\lambda y^{t+1} : \Delta_{t+1}^{\mathsf{Bool}}.\\ & x^{t+1}\ (\lambda x^t : \Delta_t^{\mathsf{Bool}}.\textsc{and}\\ & \quad (member_{t+1}\ x^t\, y^{t+1}))\ \mathit{true}\\ eq_{t+1} \equiv\ & \lambda x^{t+1} : \Delta_{t+1}^{\mathsf{Bool}}.\lambda y^{t+1} : \Delta_{t+1}^{\mathsf{Bool}}.\\ & \quad (\lambda op : \Delta_{t+1}^{\mathsf{Bool}} \rightarrow \Delta_{t+1}^{\mathsf{Bool}} \rightarrow \mathsf{Bool}.\\ & \quad \textsc{and}\ (op\ x^{t+1} y^{t+1})\,(op\ y^{t+1} x^{t+1}))\\ & \quad subset_{t+1}\end{aligned}$$

Theorem 4.2 *A formula Φ in higher-order logic over the finite base type $\mathcal{D}_1 = \{\mathsf{true}, \mathsf{false}\}$ is true if and only if its typed λ-calculus interpretation $\hat{\Phi} : \mathsf{Bool}$ is $\beta\eta$-equivalent to $\mathit{true} \equiv \lambda x : o.\lambda y : o.x : \mathsf{Bool}$. Moreover, if Φ only quantifies over universes $\mathcal{D}_i$ for $i \leq k$, then $\hat{\Phi}$ has* order *at most k, and if we also write $|\Phi|$ to denote the length of logic formula Φ, then $|\hat{\Phi}| = O(|\Phi|(2k)!)$.*

4.2 Hacking a time-bounded Turing Machine in higher-order logic

Assume that the Turing Machine we simulate has tape alphabet $\{0, 1\}$, so that the tape contents is just a big binary number. Since $|\mathcal{D}_{k+1}| = 2^{|\mathcal{D}_k|}$, it is easy to show that each element $x^{k+1} \in \mathcal{D}_{k+1}$ can be thought of as such a large integer, where the *elements* of x^{k+1} are just the *bit positions* set to 1 in its binary encoding.

If x^{k+1} codes 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. If h_1^k and h_2^k code head positions in successive IDs, then $\mathrm{succ}_k(h_1^k, h_2^k)$ expresses that the head moved one position to the right. By defining a total order $<_t$ on the

elements of $\mathcal{D}_t$, we can specify the successor succ_{t+1}; both definitions just use the intuitions found from a course in machine architecture, written down in formal logic. If the input to the Turing Machine is of length n, then any tape contents of a computation taking $\mathrm{K}_\ell(n)$ steps can be represented in $\mathcal{D}_k$ where $k = \ell + \log^* n$.

A small but suitably large $\mathcal{D}_{t+1}$ can code the finite states of the Turing Machine as an integer. Predicate zero_{t+1} defines zero in $\mathcal{D}_{t+1}$, while con^j_{t+1} identifies the constant j. To code tape contents, head position, and finite state into a single set, we need pairing and projection relations, used to define the ordered pair (a, b) as the usual (but typed) $\{\{a\}, \{a, b\}\}$. If u^{k+1}, h^k, s^k respectively code the tape contents, head position, and finite state control of a Turing Machine, we define $\mathbf{ID}(u^{k+1}, h^k, s^k, z^{k+3})$ to represent all this information in a variable z^{k+3}. 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. The relation specifying the initial configuration can then be defined as:

$$
\begin{aligned}
&\text{INITIAL}(z^{k+3}) \equiv \\
&\quad \exists u^{k+1}.\exists h^k.\exists s^k.\exists c_0^k.\exists c_1^k.\cdots\exists c_{n-1}^k. \\
&\qquad \mathbf{ID}(u^{k+1}, h^k, s^k, z^{k+3}) \wedge \\
&\qquad \text{zero}_k(h^k) \wedge \text{zero}_k(s^k) \wedge \text{zero}_k(c_0^k) \wedge \\
&\qquad \left(\bigwedge_{1 \le i \le n-1} \text{succ}_k(c_{i-1}^k, c_i^k) \right) \wedge \\
&\qquad \forall b^k.b^k \in u^{k+1} \leftrightarrow \left(\bigvee_{i \in X} b^k = c_i^k \right).
\end{aligned}
$$

The coding of $\text{INITIAL}(z^{k+3})$ is the only part of the specification with length that grows as $\Omega(n \log^* n)$. All the other components of the specification have length $O(\log^* n)$. The definitions of $\text{FINAL}(z^{k+3})$ and the transition relation δ are equally straightforward. Finally, the reflexive, transitive closure δ^* is used to iterate the transition function a sufficient number of times:

$$
\begin{aligned}
&\delta^*(u^{k+3}, v^{k+3}) \equiv \\
&\quad \exists c^{k+6}.\,[\forall (r^{k+3}, s^{k+3}). \\
&\qquad [(r^{k+3}, s^{k+3}) \in c^{k+6} \leftrightarrow [r^{k+3} = s^{k+3} \vee \\
&\qquad\quad \exists q^{k+3}.(r^{k+3}, q^{k+3}) \in c^{k+6} \wedge \\
&\qquad\quad \delta(q^{k+3}, s^{k+3})]]] \\
&\quad \wedge (u^{k+3}, v^{k+3}) \in c^{k+6}.
\end{aligned}
$$

The constant c^{k+6} codes the set of pairs (r^{k+3}, s^{k+3}) that form the transitive closure of the relation defined by δ.

5 Main results

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

Theorem 5.1 *Let M be a fixed Turing Machine that accepts or rejects an input x in $\mathrm{K}_\ell(|x|)$ steps. Then there exists a formula Φ_x in higher-order logic such that M accepts x if and only if Φ_x is true. Moreover, Φ_x only quantifies over universes $\mathcal{D}_i$ for $i \le (\log^* |x|) + \ell + 6$, and has length $O(|x| \log^* |x|)$.*

Recall that our goal was to define a formula in higher-order logic coding the decision problem, "Does Turing Machine M accept x in $\mathrm{K}_\ell(|x|)$ steps?" The formula we derived has length almost linear in $|x|$. By considering the coding of this formula in the simply-typed λ-calculus, using the translation described in Section 4.1, we derive the following corollary:

Corollary 5.2 *Let M be a fixed Turing Machine that accepts or rejects an input x in $\mathrm{K}_\ell(|x|)$ steps. Then there exists a typed closed λ-term $\hat{\Phi}_x$: Bool such that M accepts x if and only if $\hat{\Phi}_x$ reduces to true $\equiv \lambda x$: $o.\lambda y$: $o.x$: Bool. Moreover, the bound variables in $\hat{\Phi}_x$ have order $O(\log^* |x|)$, and $|\hat{\Phi}_x| = O(|x| \log^{(c)} |x|)$ for any fixed integer $c > 0$.*

Combining the last two results, we derive our most important theorem:

Theorem 5.3 (Main Theorem) *There exists a set of closed λ-terms E_n : Bool, where $|E_n| = O(n \log^{(c)} n)$ for any integer $c > 0$, such that E_n normalizes in $O(|E_n|)$ parallel β-steps, and the time needed to implement the parallel β-steps,* on any first-class machine model *[vEB90], grows as $\Omega(\mathrm{K}_\ell(n))$ for any fixed integer $\ell \ge 0$.*

The Main Theorem must apply to the "machine" defined by Lamping's algorithm, since the graph reduction operations can be easily implemented with a time 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 *There exists a set of λ-terms E_n : Bool which normalize in no more than n parallel β-steps, where the number of bookkeeping interactions that are required to normalize E_n using Lamping's graph reduction algorithm grows as $\Omega(\mathrm{K}_\ell(n))$ for any fixed integer $\ell \ge 0$.*

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 β-redexes, and thus represents a normalized λ-term, one way to read back the term is to continue graph reduction "nonoptimally," allowing forbidden duplication of λ and apply nodes. We may then ask: given a graph without β-redexes, how hard is it to read back the normal form? Recall the question "Does Turing Machine M accept x in $\mathrm{K}_{\ell+1}(|x|)$ steps?" The answer to this question is coded in the "blob" of Figure 1. As a consequence, we derive the following corollary:

Corollary 5.5 (Readback Lemma) *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 β-redexes, the λ-term coded by G_n has constant size, and the computational work required to read back the represented term grows as $\Omega(\mathrm{K}_\ell(n))$ for any fixed integer $\ell \geq 0$.*

All of the above results depend on the reduction of λ-terms where the number of ordinary β-steps dwarfs the number of parallel β-steps. We summarize this difference in our final corollary:

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

To prove the corollary, define $2'_j$ to be the fully η-expanded version of $\overline{2}\,\overline{2}\cdots\overline{2}$ consisting of j $\overline{2}$s, where $\overline{2}$ is the Church numeral for 2; note $2'_j = \Theta(2^j)$. A β-step can at most square the length of a term, so n β-reductions of a term of length m gives a term of length at most m^{2^n}. The reduction of $2'_{2\ell+2}$ to $\overline{\mathrm{K}_{2\ell+2}(1)}$ thus requires $\mathrm{K}_{2\ell}(1)$ ordinary β-steps, but only $c2^{2\ell}$ parallel β-steps for some small constant c; as a function of ℓ, certainly $\mathrm{K}_{2\ell}(1) = \Omega(\mathrm{K}_\ell(c2^{2\ell}))$.

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 $b = \Omega(2^{2^p})$.

The Main Theorem also 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, AG97]), the *times* and *par* connectives of linear logic play essentially the same role as apply and λ nodes in λ-calculus; the programming synchronization implemented by the *closure* has its counterpart in proof net *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 λ-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

We used Lamping's technology as a calculating device to prove a theorem about all possible implementations of optimal evaluation. What are Lamping's graphs doing to so cleverly encode so much sharing? In this section, we discuss two essential phenomena of these data structures: that of *superposition*, and that of *higher-order sharing*. Superposition is a coding trick where graphs for different values—for example, *true* and *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. We resist stating theorems in this section, instead encouraging the reader towards a more intuitive grasp of the finitary dynamics of graph reduction.

6.1 Superposition

In Figure 4, we see a really simple, but canonical example of superimposed graphs: the coding of *true* and *false*. Notice how the "star" sides of the sharing nodes

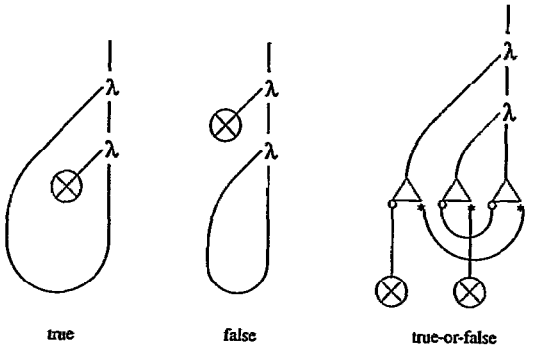


Figure 4: Superposition of true and false.

code *true*, and the "circle" sides of the sharing nodes code *false*. The two values share the λ-nodes that serve as the interface to the external context. Superposition lets us, for example, negate *both* Boolean values at the usual cost of negating *one* of them—negation just swaps the position of two wires.

This sharing graph really occurs in reducing a term—it comes from η-expanding the list $\mathbf{D}_1$ of Booleans. This is why SATISFIABILITY can be coded in a sub-exponential number of parallel β-steps: every row of the truth table calculation is superimposed into a single, "shared" row. The truth values for every row are computed all at once! Superposition says $\mathbf{P} = \mathbf{NP}$, among other falsehoods, if sharing is free.

Another beautiful example comes from Church numerals. Superimposition can create a numeral that is 2 and 3 at the same time. Unlike Booleans, there are lots of numerals, and an arbitrarily large finite number of them can be superimposed. We can, for example, square an *unbounded* number of numerals in the time it usually takes to square *one* numeral, where the style of computation looks like a λ-calculus optimal evalution implementation of SIMD (single instruction, multiple data) parallelism. The sharing network for each numeral is like a separate processor, serving as multiple data; the λ- and apply-nodes serve as interface to the external context; the multiplexors and demultiplexors replicate and pump the single-instruction stream (the code for, say, the squaring function) to each of the processors. The real work of the computation becomes *communication*, performing the η-expansion on the lists, and communicating the function to different processors.

6.2 Higher-order sharing

Higher-order sharing constructs 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. The key idea 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." A network made of $2k$ sharing nodes can then implement a path of length 2^k, where the naive *stack semantics* implements binary counting.

However, this construction merely shares an *application* node; to get a truly powerful network, we need to be able to share *sharing nodes* as well. The construction is inductive; the basis is described by the simple network of Figure 5, 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 previously to share a function application, can instead be used to share a sharing node, as illustrated in Figure 6. We can then go one step further, considering a *nested* construction, where we share the sharing of sharing nodes, *ad nauseam*. Figure 7 hints at how such a network of 2^ℓ 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* (a static form of *indices*) $0 \le \ell \le j$. 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 $\overline{N}$ to mean N with

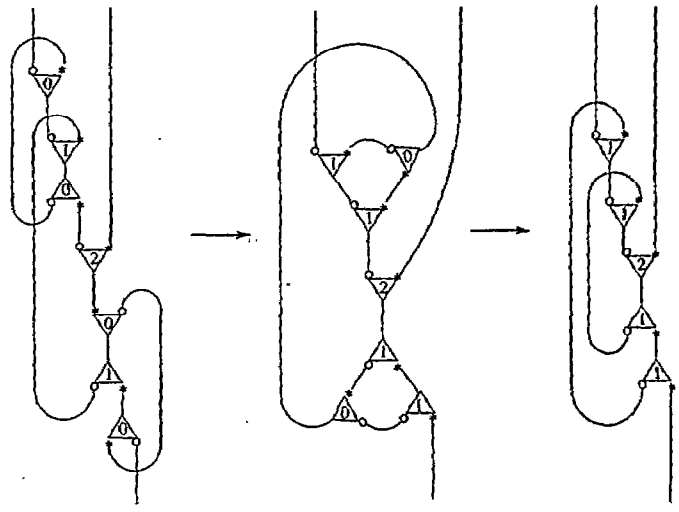


Figure 7: A "nested" construction.

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, $\overline{N_j}$, 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 Figure 8).

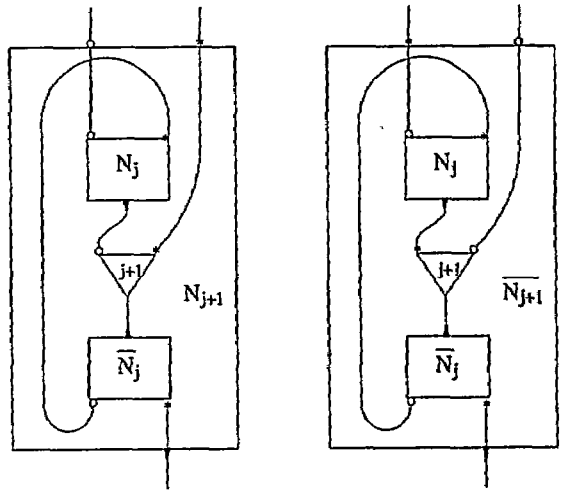


Figure 8: The generic construction of an elementary network.

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.

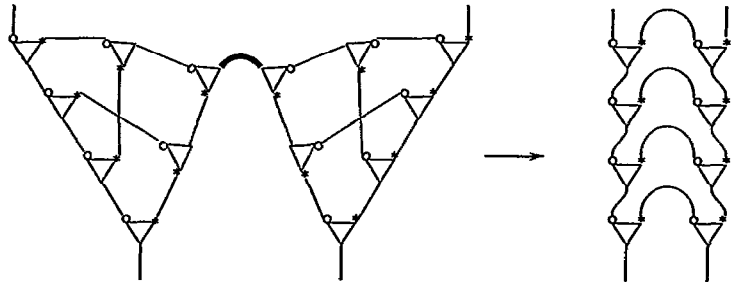


Figure 5: A simple sharing network.

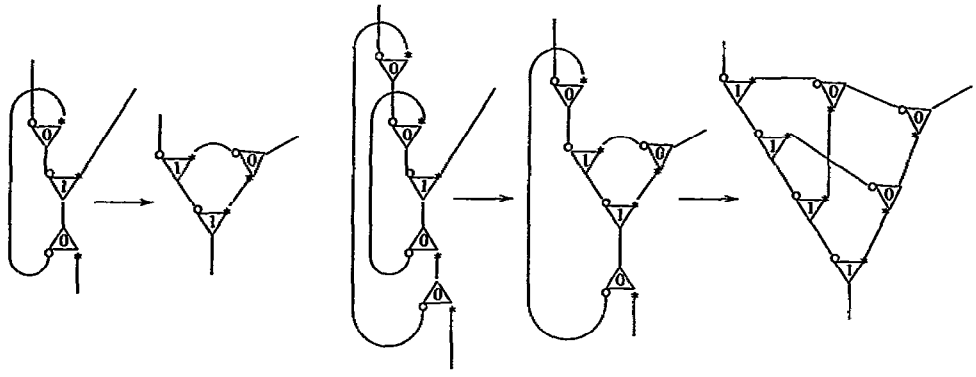


Figure 6: How to share a sharing node.

Lemma 6.1 *The network N_j (respectively, $\overline{N_j}$) normalizes to a complete binary tree with $\mathrm{K}_j(1)$ 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 Figure 8.*

This kind of sharing network results from the parallel β-reduction of the η-expanded term $2_j \equiv \overline{2}\,\overline{2} \cdots \overline{2}$ of j $\overline{2}$s, 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 β-steps to the Church numeral for $\mathrm{K}_j(1)$. The exponential length is sufficient to code a copy of N_j and $\overline{N_j}$; after normalization, these networks expand to construct $\mathrm{K}_j(1)$ 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 that the cost of implementing a sequence of n parallel β-steps in the reduction of a λ-term is not bounded by any Kalmár-elementary function $\mathrm{K}_\ell(n)$. This theorem clearly indicates that the parallel β-step is not a unit-cost operation. As a consequence, cost models based solely on this metric are unrealistic, and implementations designed to minimize parallel β-steps run the risk of being seriously misguided. While intuitive sentiments of this sort may have been longstanding in the functional programming community, our results provide a precise, mathematical analysis of the relevant computational phenomena, thus justifying such heretofore unsubstantiated intuitions.

Given that the parallel β-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? The study of optimal reduction has provided important insights and connections between linear logic, control structures, context semantics and the geometry of interaction, and intensional semantics, with hope of applying its ideas as well in the areas of full abstraction and game semantics. 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?

We believe that the data structures and algorithms underlying Lamping's algorithm provide genuinely new implementation technology that deserve consideration in the world of pragmatics. This technology deserves serious consideration for providing a novel semantics, while simultanously providing the promise of efficient implementation. Optimal evaluation provides a new, hybrid procedure-calling paradigm, encompassing the virtues of both call-by-value and call-by-name. The usual trinity of call by name, value, and reference re-

veals little about the computational world around us, and a great deal about our limited knowledge of that universe.

With regard to efficiency considerations, we wish to caution against undue and unjustified pessimistic conclusions that may be drawn from our analysis. Recall that 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. *There is no cheap way to implement a generic simulation*— in our analysis, either the parallel β-step had to be expensive, or there had to be many such steps. Computation is expensive.

Minimizing the number of β-steps via sharing subtly misses the point of designing pragmatically efficient implementations. Far more relevant is the efficiency of the technology that *implements* sharing. It is here that lies the promise of optimal evaluation technology. The importance of Lamping's graph reduction algorithm is its attempt to minimize the *cost of substitution* by maximizing the dual *sharing* of values and continuations. In particular, subexpressions with the same *Lévy label* are almost always shared. It can be shown that the number of interactions of λ nodes, apply nodes, and sharing nodes in optimal graph reduction is bounded by a polynomial in the number of *unique* labels generated in a reduction [LM97]. If we believe that a label identifies a set of similar subexpressions, this result means that the graph reduction algorithm is maximizing sharing up to a polynomial factor. We would like to understand better how this computational work compares to that found in more conventional interpreters.

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 *bracket* and *croissant* interactions can be greatly reduced, or whether they are essential, even with the optimization possible from so-called "safe nodes." Careful comparisons are needed between optimal graph reduction and conventional methods of β-reduction, beginning with standard call-by-name technology, as they are both correct strategies; it remains to be seen whether optimal graph reduction can be made competitive with call-by-value—and if so, it would certainly be superior, since it has a greater functionality.

In summary, graph reduction for optimal evaluation presents a new technology for language implementation. 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 would like to say the same thing.

Acknowledgements. For many relevant and fruitful discussions, suggestions, corrections, admonishments, and encouragements, we thank Arvind, Stefano Guerrini, Paris Kanellakis, Jakov Kucan, Julia Lawall, Simone Martini, Albert Meyer, Luca Roversi, and René Vestergaard.

References

[Asp96] Andrea Asperti. *On the complexity of beta-reduction.* 1996 ACM Symposium on Principles of Programming Languages, pp. 110–118.

[AG97] Andrea Asperti and Stefano Guerrini. The Optimal Implementation of Functional Programming Languages. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press. To appear.

[vEB90] Peter van Emde Boas. *Machine models and simulation.* Handbook of Theoretical Computer Science, volume A, pp. 1–66. North Holland, 1990.

[Dys79] Freeman Dyson. Disturbing the Universe. Harper and Row, 1979.

[FS91] Gudmund S. Frandsen and Carl Sturtivant. *What is an efficient implementation of the λ-calculus?* 1991 ACM Conference on Functional Programming and Computer Architecture (J. Hughes, ed.), LNCS 523, pp. 289–312.

[Gan69] Robin Gandy. *An early proof of normalization by A. M. Turing.* To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism, (J. P. Seldin and J. R. Hindley, eds.), pp. 453–455. Academic Press, 1980.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. Proofs and Types. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1989.

[GAL92a] Georges Gonthier, Martin Abadi, and Jean-Jacques Lévy. *The geometry of optimal lambda reduction.* 1992 ACM Symposium on Principles of Programming Languages, pp. 15–26.

[GAL92b] Georges Gonthier, Martin Abadi, and Jean-Jacques Lévy. *Linear logic without boxes.* 1992 IEEE Symposium on Logic in Computer Science, pp. 223–234.

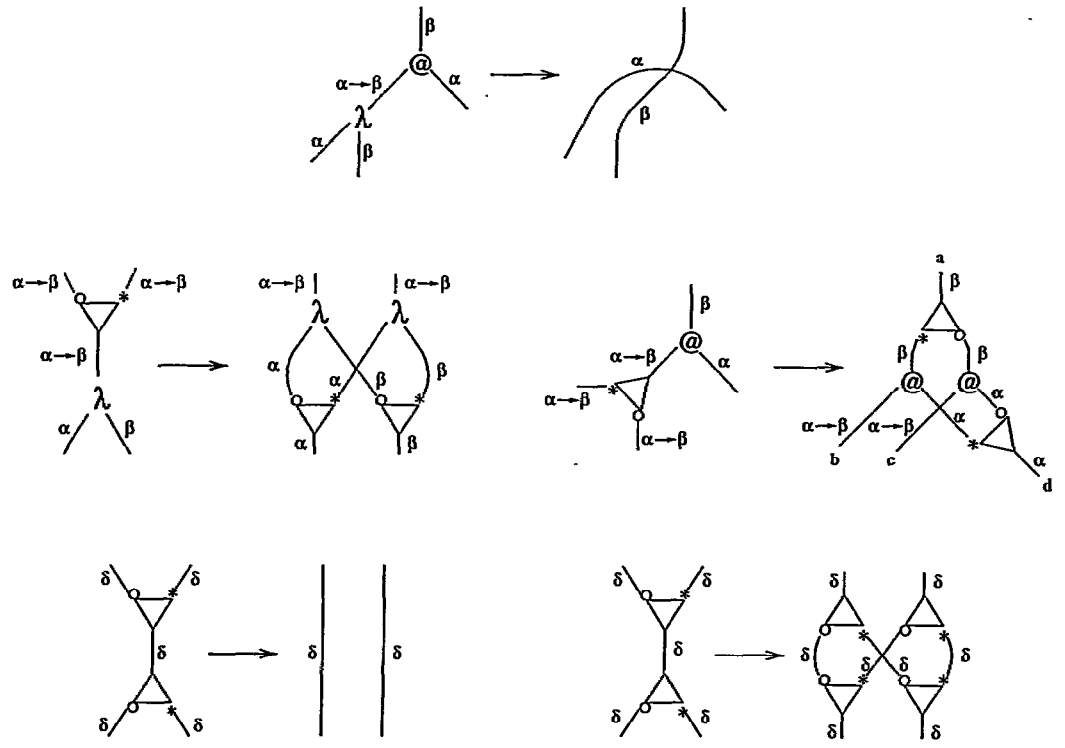
[Gue96] Stefano Guerrini. *Theoretical and practical issues of optimal implementations of functional languages.* Ph. D. Thesis, Università di Pisa, 1996.

[HS87] J. Roger Hindley and Jonathan P. Seldin. Introduction to Combinators and Lambda Calculus. Cambridge University Press, 1987.

[Hoa85] Charles Antony Richard Hoare. The Mathematics of Programming. Inaugural Lecture, Oxford University. Clarendon Press, October 1985.

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

Appendix A: Graph reduction rules



[Kal43] László Kalmár. *Egyszerű példa eldönthetetlen aritmetikai problémára (Ein einfaches Beispiel für ein unentscheidbares arithmetisches Problem).* Matematikai és fizikai lapok 50, 1943, pp. 1–23. Hungarian with German abstract.

[Kat90] Vinod Kathail. *Optimal interpreters for lambda-calculus based functional languages.* Ph.D. Thesis, MIT, May 1990.

[Laf95] Yves Lafont. *From proof nets to interaction nets.* Advances in Linear Logic (J.-Y. Girard, Y. Lafont, L. Regnier, eds.), Cambridge University Press, 1995.

[Lam90] John Lamping. *An algorithm for optimal lambda calculus reduction.* 1990 ACM Symposium on Principles of Programming Languages, pp. 16–30.

[LM96] Julia L. Lawall and Harry G. Mairson. *Optimality and inefficiency: what isn't a cost model of the lambda calculus?* 1996 ACM International Conference on Functional Programming, pp. 92–101.

[LM97] Julia L. Lawall and Harry G. Mairson. *On the global dynamics of optimal graph reduction.* 1997 ACM International Conference on Functional Programming, pp. 188–195.

[Lévy78] Jean-Jacques Lévy. *Réductions correctes et optimales dans le lambda-calcul.* Thèse d'Etat, Université Paris 7, 1978.

[Lévy80] Jean-Jacques Lévy. *Optimal reductions in the lambda-calculus.* To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism, (J. P. Seldin and J. R. Hindley, eds.), pp. 159–191. Academic Press, 1980.

[Mai92] Harry G. Mairson. *A simple proof of a theorem of Statman.* Theoretical Computer Science 103 (1992), pp. 387–394.

[Mey74] Albert R. Meyer. *The inherent computational complexity of theories of ordered sets.* Proceedings of the International Congress of Mathematicians, 1974, pp. 477–482.

[Sch94] Silvan S. Schweber. QED and the men who made it: Dyson, Feynman, Schwinger, and Tomonaga. Princeton University Press, 1994.

[Sta79] Richard Statman. *The typed λ-calculus is not elementary recursive.* Theoretical Computer Science 9, 1979, pp. 73–81.

[Tai67] William W. Tait. *Intensional interpretation of functionals of finite type I.* J. Symbolic Logic 32, pp. 198–212, 1967.

[Tri78] Myron Tribus. *Thirty years of information theory.* The Maximum Entropy Formalism (ed. R. D. Levine and M. Tribus), pp. 1–14. MIT Press, 1978.

[Vui74] Jean Vuillemin. *Correct and optimal implementation of recursion in a simple programming language.* J. Computer and System Sciences 9:3, pp. 332-354.