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
Boston University
Boston, Massachusetts 02215
mairson@cs.bu.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, $O(\mathbf{K}_\ell(n))$ where $\mathbf{K}_\ell(n)$ 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.

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

[*]Supported by CONFER.
[†]Supported by ONR Grant N00014-93-1-1015, NSF Grant CCR-9619638, and the Tyson Foundation.

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 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 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, Abadi, and Lévy [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 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, $O(f(n))$ where $f(n)$ is any fixed stack of 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 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. 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* $\mathbf{K}_\ell(n)$ as $\mathbf{K}_0(n) = n$, and $\mathbf{K}_{t+1}(n) = 2^{\mathbf{K}_t(n)}$ [Kal43].

Main Theorem. *Let $\ell \geq 0$ be any fixed integer. Then there exists a set of explicitly typed, closed λ-terms $E_n : \mathsf{Bool}$, where $|E_n| = O(n)$, 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 $\Omega(\mathbf{K}_\ell(n))$.*

Proof Sketch. For a fixed Turing Machine M, and input x of length n, consider the question, "Does M accept x in $\mathbf{K}_{\ell+1}(n)$ 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 but linear 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 $\mathbf{K}_\ell(|E|)$; we would then have shown that

$$\text{DTIME}[\mathbf{K}_{\ell+1}(|x|)] \;\subseteq\; \text{DTIME}[\mathbf{K}_\ell(|E|)].$$

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* $\mathbf{K}_\ell(n)$ 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 β-steps is proven by computing

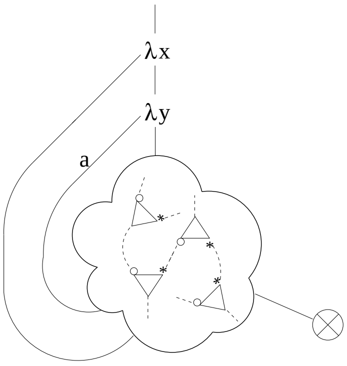


Figure 1: The blob.

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 $\mathbf{K}_\ell(n)$ steps,* no matter how we choose to assign the work associated with this decision problem to the individual parallel-β steps. □

[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.

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 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 analogue 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 β-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 β-step, but rather by the ancillary methodology which facilitates that operation. In particular, we identify the computational paradigms of *superposition* of values[2] 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 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.

The material in this paper is entirely self-contained and from first principles. 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—these comprising the essence of the theorems of Statman and Meyer [Sta79, Mey74], as fundamentally reconstructed in [Mai92]. Section 5 contains the main results of the paper. Finally, for those interested in the algorithmics of Lamping's technology, Section 6 describes the basic graph constructions involving sharing nodes that allow huge computations to be simulated by so few parallel β-steps.

[2] 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 λ-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 $(\lambda f.\lambda z.z(fM)(fN))(\lambda x.Fx)$, 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.

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 does 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 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 the optimal graph reduction technique, a λ-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 λ-nodes that represent their respective binders. Since we work in a typed setting, we shall label each wire edge of the graph with a suitable type; for brevity, we shall usually use the notation β^α instead of $\alpha \to \beta$. We emphasize that 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.

For instance, the graph in Figure 5.(1) is the initial representation of the λ-term $M = (\overline{2}_{o\to o}\ \overline{2}_o)$, where $\overline{2}_\alpha = \lambda s : \alpha \to \alpha.\lambda z : \alpha.s(sz) : (\alpha \to \alpha) \to \alpha \to \alpha$. The type $(\alpha \to \alpha) \to \alpha \to \alpha$ is the type of Church numerals (or iterators) for α, and is usually denoted $\mathbf{N}_\alpha$.

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 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 port where

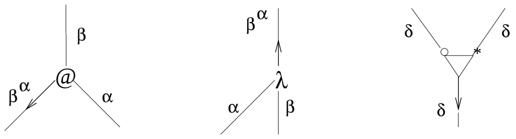


Figure 2: Nodes, ports and their types.

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, 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.

We call 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 $(\overline{2}_{o\to o}\,\overline{2}_o)$. 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 didactical criteria, and occasionally for graphical convenience.

The most important of the graph rewriting rules is β-reduction, where $(\lambda x.M)N$ is replaced by $M[N/x]$; see Figure 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 β-reduction can be expressed by the completely local graph rewriting rule in Figure 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 β-redex in $(\overline{2}_{o\to o}\,\overline{2}_o)$, we derive the graph in Figure 5.(2). The next redex involves a shared λ-expression in the function position. In ordinary graph reduction, the entire representation of the function would be duplicated. In contrast, optimal graph reduction technique proceeds in a lazy fashion, duplicating the external λ node, but still sharing its body; ee Figure 5.(3). Since the binder has been duplicated, thus allowing

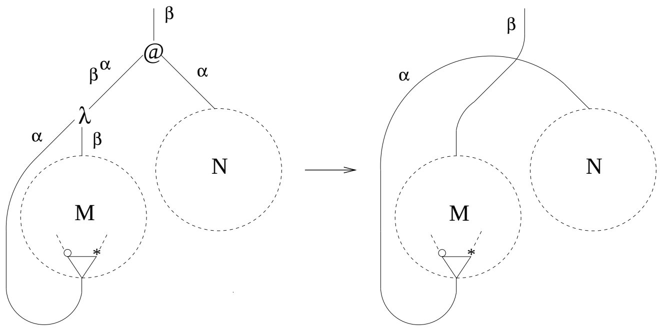


Figure 3: β-reduction

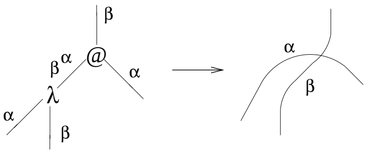


Figure 4: the β-rule

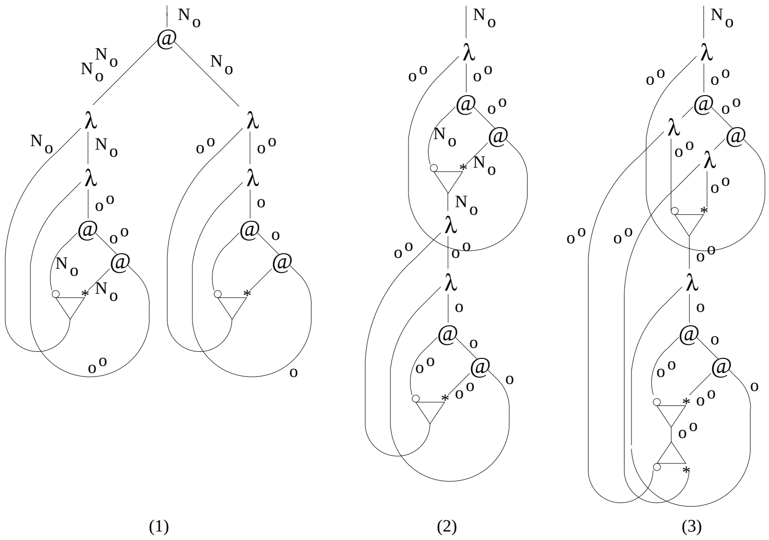


Figure 5: Graph reduction of $(\overline{2}_{o\to o}\ \overline{2}_{o})$

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)

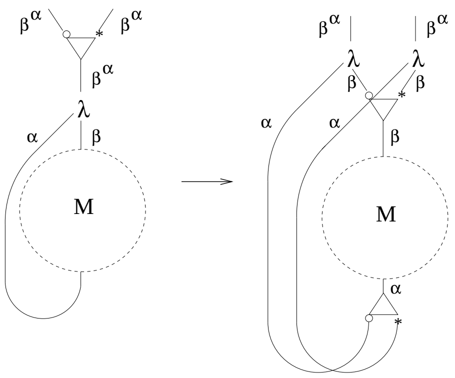


Figure 6: the duplication of abstractions

and the unsharing (fan-out) come in pairs. Although 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 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 labelling 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 (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 λ, as described in Figure 7. Note that the type of fan nodes

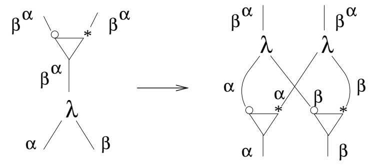


Figure 7: Fan-λ interaction

(that is, the type of their principal port) may only decrease by the firing of this rule.

Next, by applying the fan-λ interaction rule, we get the graph in Figure 5.(3). Two β-redexes have thus been created; by firing each of them, we derive the graph in Figure 8.(4). Then we fire the fan-λ rule, obtaining the graph in Figure 8.(5). There are no more β-redexes in this graph, as well as no fan-λ interactions, but we must proceed in the duplication process with considerable caution nonetheless. In particular, the graph rewriting rule shown in Figure 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 optimality.

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 Figure 8.(5) are not "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 $\overline{2}_{o\rightarrow o}$, in the process of duplicating $\overline{2}_o$—they must duplicate each other, according to the rule in Figure 10.(2). Note that the type of fan-nodes is preserved by this interaction. Now (see Figure 8.(6)), we have a fan-out in front of the function-port of the application. In this case, we can apply the rule in Figure 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-λ interaction, the type of the sharing node strictly decreases.

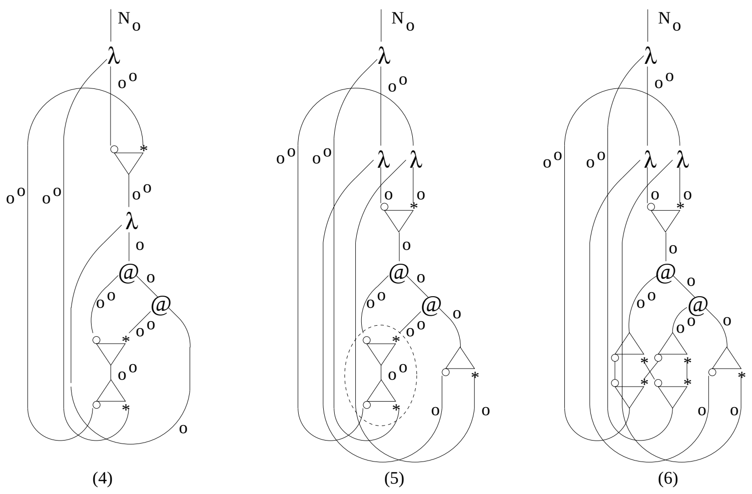


Figure 8: Graph reduction of $(\overline{2}_{o\to o}\ \overline{2}_o)$

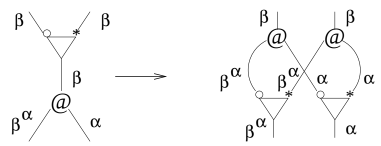


Figure 9: not optimal duplication of the application

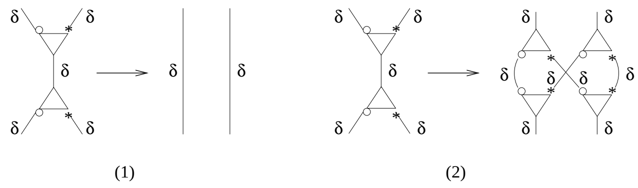


Figure 10: fan-annihilation rule

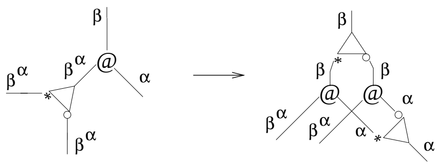


Figure 11: fan-apply interaction

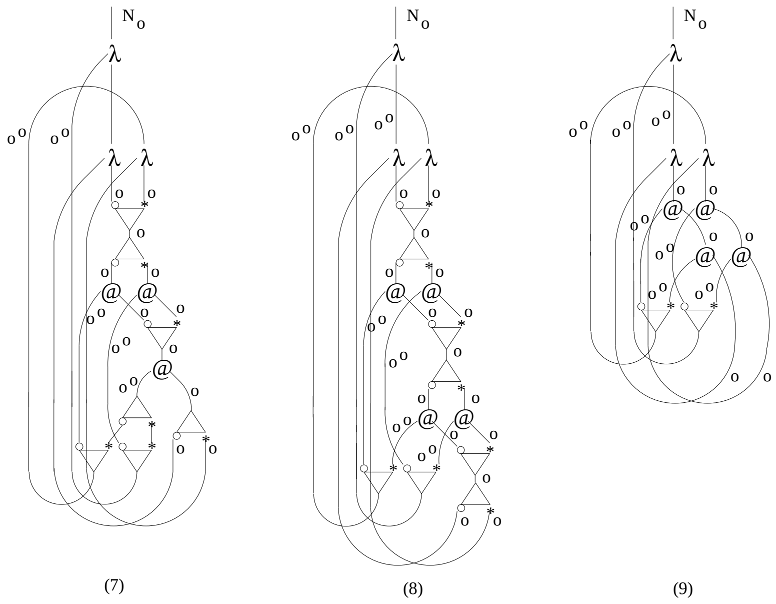


Figure 12: Graph reduction of $(\overline{2}_{o\to o}\ \overline{2}_o)$

By firing this rule twice, we derive the graphs in Figure 12.(8), 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 Figure 10.(1).

By three applications of the fan-annihilation rule, we derive the graph in Figure 12.(9). The three last steps are, respectively, a fan-λ rule, and two β-reductions; see Figure 13. The graph in Figure 13.(12) is in normal form with respect to Lamping's algorithm; we urge the reader to understand why this graph represents the Church integer $\overline{4}_o$ 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 ($\star$ or $\circ$) that the paired fan-in node was entered.

3 The η-expansion method

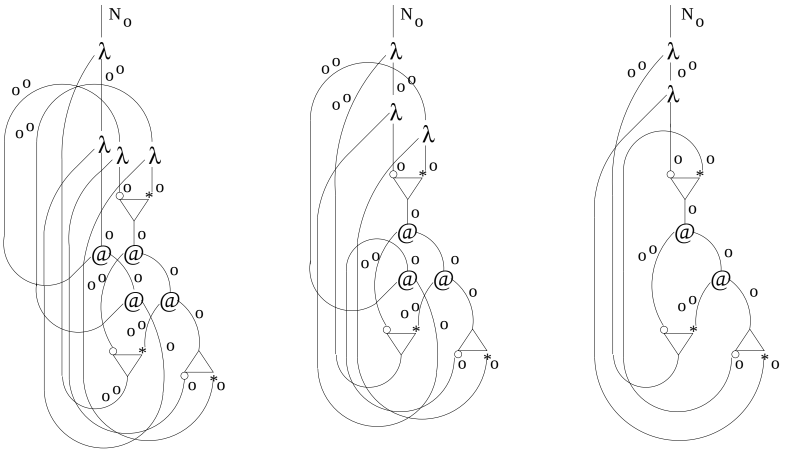
Given a simply typed λ-term E, we show how to construct a variant E' that is $\beta\eta$-equivalent to E, derived by introducing *η-expansions* of bound variables in E. The size of E', and the size of the initial sharing graph that represents E', are larger than E by only a small constant factor. Moreover, the number of parallel 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. In order to make these calculations more precise, we need to define what we mean by the *size* of types, λ-terms, and graphs:

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

$$\begin{aligned}\|o\| &= 1\\ \|\alpha\to\beta\| &= 1+\|\alpha\|+\|\beta\|.\end{aligned}$$

Similarly, we inductively define the size of a simply-typed term *as:*

$$\begin{aligned}|x| &= \|\sigma\| \qquad \text{if } x \text{ has type } \sigma\\ |\lambda x:\sigma.E| &= 1+|E|\\ |EF| &= 1+|E|+|F|.\end{aligned}$$



(10) (11) (12)

Figure 13: Graph reduction of $(\overline{2}_{o\to o}\ \overline{2}_o)$

The number $|E|$ simply counts 1 for each λ and apply in E, plus $\|\sigma\|$ for each variable of type σ. 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 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. This technique leads to an efficient *accounting mechanism* which bounds the duplication of λ and apply nodes in a graph reduction, and hence bounds the number of parallel β-steps. In addition, it serves as a lovely *optimization* method, where parallel β-reduction is simulated by the interaction between sharing nodes *only.*

Definition 3.2 *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 follows:*

$$\begin{aligned}\eta_o(x) &= x\\ \eta_{\alpha_1\to\cdots\to\alpha_k\to o}(x) &= \lambda y_1 : \alpha_1.\ \ldots\ \lambda y_k : \alpha_k.x\ (\eta_{\alpha_1}(y_1))\ldots\ (\eta_{\alpha_k}(y_k))\end{aligned}$$

In the graph representation, each η-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 Figure 14). When the variable is of base type o, this subgraph is just a wire. The graph representing an η-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 σ. Notice that if $\sigma \equiv \alpha_1 \to \alpha_2 \to \cdots\alpha_k \to o$, then $[\![G]\!] = 2k + \sum_{1\leq i\leq k}[\![G_{\alpha_i}]\!]$, where G_{α_i} represents $\eta_{\alpha_i}(y_i)$. $\square$

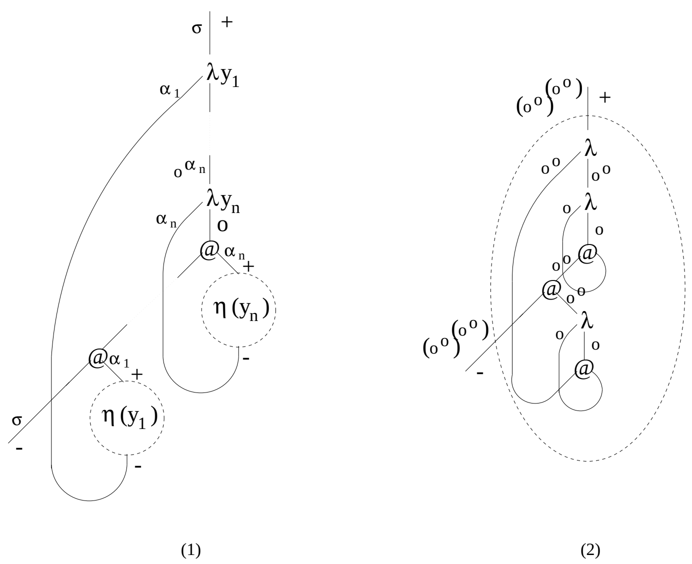


Figure 14: (1) η-expansion; (2) $\eta_{(o\to o)\to o\to o}(x)$.

Lemma 3.4 *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*[3] *are at base type, and $[\![\Delta(x)]\!] \le 3\|\sigma\|$.*

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

Suppose $\sigma = \alpha_1 \to \ldots \to \alpha_n \to o$. Then the graph G coding $\eta_\sigma(x)$ has the structure depicted in Figure 14(1), 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 $\eta_{\alpha_i}(y_i)$; see Figure 15.(1). 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 $\eta_{\alpha_i}(y_i)$; see Figure 15.(2). The lemma follows by induction on the α_i. $\Box$

Example 3.5 The graph in Figure 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. $\Box$

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 $\Delta^t(x)$ 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 $[\![\Delta^t(x)]\!] \le (2+t)\|\sigma\|$.*

In the initial sharing graph coding a λ-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 λ-node that represents the binding. Because this parameter port is not a primary port, no interaction is possible between the sharing nodes and the λ-node; the sharing nodes are "stuck" until the λ-node is annihilated in a parallel β-reduction. We seek to control the possible node replication that could result from such a reduction, by forcing the λ-term $\lambda x : \sigma.E$ to be applied to $\eta_\sigma(x')$. We may then conclude that the duplication caused by a

[3] We chose the notation $\Delta(x)$ to remind the reader that the graph is defined by propagating the sharing nodes (hence the Δ) to base type.

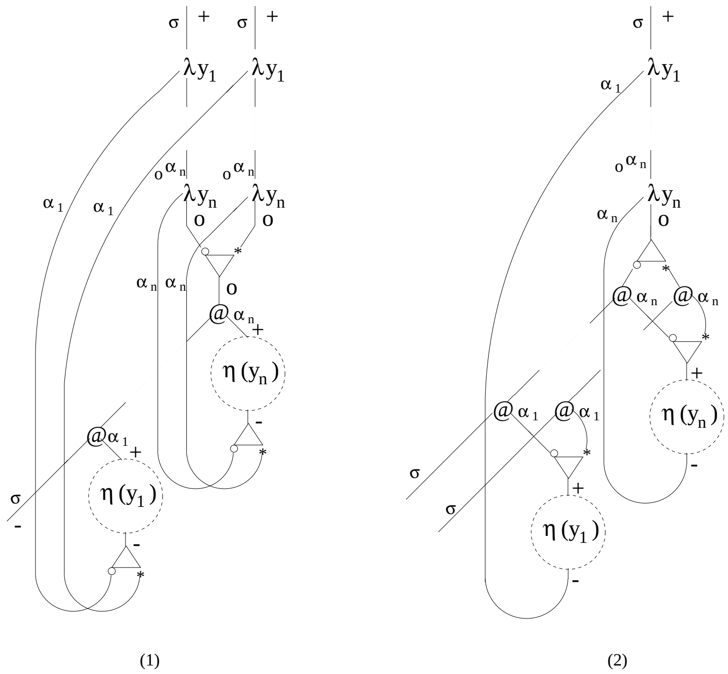


Figure 15: Fan propagation inside $\eta(x)$.

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 in 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 λ-term. The* optimal root $\mathbf{or}(M)$ *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.

It should be clear that the transformation is applied at most once to any subterm $\lambda x : \sigma.E$. Since $\mathbf{or}(M)$ is obtained by M by means of η and β-expansions, we also know that $M =_{\beta\eta} \mathbf{or}(M)$. The transformation can be understood as duplicating, 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 3.8 *We define $\Delta(M)$ to be the sharing graph obtained from* $\mathbf{or}(M)$ *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 β-reduction of all preliminary redexes, all sharing nodes are positioned to interact at the positive entry of the subgraphs representing η-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 λ-term if it is at base type, or occurs at most once. A typed λ-term is* trivial *if all of its bound variables are trivial.*

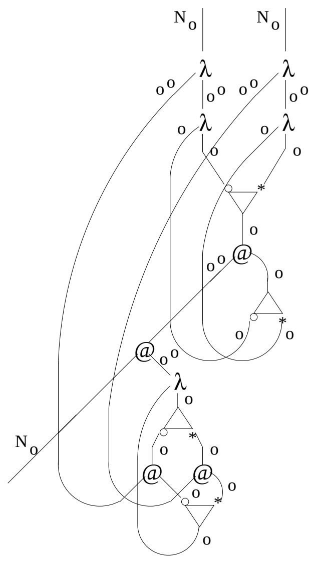


Figure 16: Duplication of $\eta_{(o\to o)\to o\to o}(x)$.

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

Proof. Let G be the sharing graph coding $\mathbf{or}(F) \equiv \lambda x':\sigma.F(\eta_\sigma(x'))$, and let G' be derived by graph reduction of the outermost β-redex, representing the term $\lambda x':\sigma.E[\eta_\sigma(x')/x]$ (see Figure 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 λ and apply in E, it should be clear that $[\![G_E]\!] \le |E| - t\|\sigma\|$.

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

$$[\![\Delta(F)]\!] \;\le\; 1 + (|E| - t\|\sigma\|) + (1+t)\|\sigma\| \;=\; 1 + |E| + \|\sigma\| \;=\; |F| + \|\sigma\|.$$

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

Theorem 3.12 *Let F be a simply-typed λ-term. Then $[\![\Delta(F)]\!] \le 2|F|$.*

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 $\eta_\alpha(x)$ over variables x of type α. Assume without loss of generality that the name of each bound variable is unique; then by Lemma 3.11,

$$\begin{aligned}
[\![\Delta(F)]\!] \;&\le\; \left(|F| - \sum_{\substack{x\,:\,\sigma \\ x \text{ nontrivial}}} \mu(x)\|\sigma\|\right) + \left(\sum_{\substack{x\,:\,\sigma \\ x \text{ nontrivial}}} (1+\mu(x))\|\sigma\|\right) \\
&\le\; |F| + \left(\sum_{\substack{x\,:\,\sigma \\ x \text{ nontrivial}}} \|\sigma\|\right),
\end{aligned}$$

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

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 σ contributes $\|\sigma\|$ to the sum. Suppose we had instead chosen the definition of size as

$$\begin{aligned}\Lambda(x) &= 1\\ \Lambda(\lambda x:\sigma.E) &= 1+\|\sigma\|+\Lambda(E)\\ \Lambda(EF) &= 1+\Lambda(E)+\Lambda(F),\end{aligned}$$

so that $\Lambda(E)$ is the length of the explicitly-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.

Theorem 3.13 *The total number of β-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 λ-node can be created during the reduction. Since β-reduction can only make the graph *smaller* via annihilation of complementary λ and apply nodes, the total number of β-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 all 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 $\mathbf{or}(M)$ is its *strong normalization*, as well as the strong normalization of M. This observation is a trivial corollary of a result due to Lévy ([Lévy78], Theorem 4.4.6):

Theorem 3.14 *Let Σ be any finite (possibly parallel) reduction of a term M. Then any reduction Σ'* relative[4] *to Σ is terminating.*

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

Theorem 3.15 *The simply typed λ-calculus is strongly normalizing.*

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

[4] 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., p. 64).

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.*[5] 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. The reason is simple: were a parallel β-step implemented in unit cost, we would have shown that PTIME equals $\mathrm{DTIME}[2^n]$, 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 λ-term, and that term normalizes in a polynomial number of parallel β-steps 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.[7]

In fact, for any integer $\ell \geq 0$, we can construct generic reductions of this kind for $\mathrm{DTIME}[\mathbf{K}_\ell(n)]$. 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* $\mathbf{K}_\ell(n)$. Observe that were the cost contained in $O(\mathbf{K}_\ell(n))$, then by simulating arbitrary computations in $\mathrm{DTIME}[\mathbf{K}_{\ell+1}(n)]$, and requiring only polynomially many β-steps (via the η-expansion method) at cost $O(\mathbf{K}_\ell(p(n)))$ for some polynomial p, we would have shown that $\mathrm{DTIME}[\mathbf{K}_{\ell+1}(n)]$ is contained in $\mathrm{DTIME}[\mathbf{K}_\ell(p(n))]$, which is again contradicted by the time hierarchy theorem.

4.1 Deciding truth of formulas in higher-order logic

Rather than code directly in the simply-typed λ-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 λ-calculus. The complexity of deciding truth for this logic was originally analyzed by Meyer, and the relation of this analysis to typed λ-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 $\mathcal{D}_{t+1} = \mathit{powerset}(\mathcal{D}_t)$. We analyze the complexity of deciding the the truth of formulas with quantification over elements of $\mathcal{D}_t$, for any $t \geq 0$, 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 $a <_0 b$, $a = b$, $a \in y^1$ (where a and b are either constants representing elements of $\mathcal{D}_0$, or variables ranging over elements of $\mathcal{D}_0$), and $x^t \in y^{t+1}$. 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?

4.2 Primitive recursive iteration in higher types

The truth of formulas in higher-order logic can be decided by compiling them into short typed λ-terms, and 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 $\{x_1, x_2, \ldots, x_\ell\}$ be a set of λ-terms, each of first-order type α; then

$$L \equiv \lambda c : \alpha \to \tau \to \tau . \lambda n : \tau . c\ x_1\ (c\ x_2\ \ldots\ (c\ x_\ell\, n)\ \ldots)$$

is a λ-term of type $(\alpha \to \tau \to \tau) \to \tau \to \tau$, for *any* type τ. We abbreviate this list construction as $[x_1, x_2, \ldots, x_\ell]$; observe that the variables c and n abstract over the list constructors **cons** and **nil**. In the simply typed λ-calculus,

[5] This *reduction* is just a compiler that takes an arbitrary Turing Machine M running in some time bound $t(n)$ on an input x of size n, and produces a typed λ-term e : Bool such that e reduces to the term coding **true** iff M accepts x in $t(n)$ steps. The "logspace" means that the compiler has only $O(\log n)$ bits[6] of internal memory to carry out the compilation, ensuring that the output has length polynomial in n.

[7] 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.

list iteration can be used to implement primitive recursion. For example, given λ-terms **succ** and **0** for zero and successor on Church numerals, the length of a list of terms of type α can be computed by

$$\mathit{length} \equiv \lambda L : (\alpha \to \mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int} \to \mathsf{Int}.L\ (\lambda x : \alpha.\mathbf{succ})\ \mathbf{0},$$

where $\mathsf{Int} \equiv\ (\nu \to \nu) \to \nu \to \nu$, and τ is set to Int in the above definition of L.

List iteration is ideal for realizing quantifier elimination: 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 $\Delta_t \to \mathsf{Bool}$. Then the truth of $\forall x^t.\Phi(x^t)$ can be coded as the λ-term $\mathbf{D}_t\ (\lambda x^t : \Delta_t.\text{AND}\ (\hat{\Phi}\, x^t))\ \mathit{true}$, and the truth of $\exists x^t.\Phi(x^t)$ can be coded as the λ-term $\mathbf{D}_t\ (\lambda x^t : \Delta_t.\text{OR}\ (\hat{\Phi}\, x^t))\ \mathit{false}$, where AND, OR, *true* and *false* are λ-terms coding up Boolean logic.[8] Observe, for example, that the latter reduces to $\text{OR}\ (\hat{\Phi}\, e_1)\, (\text{OR}\ (\hat{\Phi}\, e_2)\ \ldots\ (\text{OR}\ (\hat{\Phi}\, e_k)\, \mathit{false})\ \ldots)$, where e_j is a λ-term coding the jth element of $\mathcal{D}_t$, $1 \le j \le k = |\mathcal{D}_t|$. As we will see, the prime formulas can also be simulated using list iteration.

4.3 Coding elements of the type hierarchy

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

$$\begin{array}{rcl}
\mathit{true} & \equiv & \lambda t : o.\lambda f : o.t : \mathsf{Bool} \\
\mathit{false} & \equiv & \lambda t : o.\lambda f : o.f : \mathsf{Bool} \\
\text{AND} & \equiv & \lambda p : \mathsf{Bool}.\lambda q : \mathsf{Bool}.\lambda t : o.\lambda f : o.p\,(q t f)\, f : \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} \\
\text{OR} & \equiv & \lambda p : \mathsf{Bool}.\lambda q : \mathsf{Bool}.\lambda t : o.\lambda f : o.p\, t\, (q t f) : \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} \\
\text{NOT} & \equiv & \lambda p : \mathsf{Bool}.\lambda t : o.\lambda f : o.p\, f\, t : \mathsf{Bool} \to \mathsf{Bool} \\
\text{IF} & \equiv & \lambda p : \mathsf{Bool}.\lambda q : \mathsf{Bool}.\text{OR}\ (\text{NOT}\ p)\, q : \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}
\end{array}$$

Projection functions represent the elements of $\mathcal{D}_0$, where i is coded by e_i:

$$\mathsf{e}_i \equiv \lambda x_0 : o.\lambda x_1 : o.\cdots \lambda x_{b-1} : o.x_i : \aleph \equiv \overbrace{o \to o \to \cdots \to o}^{b \text{ times}} \to o$$

Using these projection functions, the total order $<_0$ on $\mathcal{D}_0$ is defined by tabulation:

$$\begin{array}{rcl}
\mathit{less}_0 & \equiv & \lambda x : \aleph.\lambda y : \aleph. \\
& & \quad \lambda t : o.\lambda f : o. \\
& & \quad\quad x\ (y f t t \cdots t)\, (y f f t \cdots t) \cdots (y f f f \cdots f) \\
& & \quad\quad\quad : \aleph \to \aleph \to \mathsf{Bool}
\end{array}$$

Then the set $\mathcal{D}_0$ can be represented as the list $\mathbf{D}_0$ of its coded elements:

$$\mathbf{D}_0 \equiv \lambda c : \aleph \to \tau \to \tau.\lambda n : \tau.c\ \mathsf{e}_0\, (c\ \mathsf{e}_1\ \cdots\ (c\ \mathsf{e}_{b-1}\, n)) :\ (\aleph \to \tau \to \tau) \to \tau \to \tau$$

We abbreviate the type of $\mathbf{D}_0$ as Δ_0; in general, let $\Delta_{k+1} \equiv \Delta_k^*$, where for any type α, we define $\alpha^* \equiv (\alpha \to \tau \to \tau) \to \tau \to \tau$, and $\Delta_0 \equiv \aleph$. Recall that the *order* of a type is a measure of its higher-order functionality, where $\text{order}(o) = 0$ and $\text{order}(\alpha \to \beta) = \max\{1 + \text{order}(\alpha), \text{order}(\beta)\}$; thus the order of $\Delta_0[\tau := \mathsf{Bool}]$ is 3, and $\Delta_j[\tau := \mathsf{Bool}]$ is $2j + 3$.

Next, for each integer $t > 0$, 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 define a term *double* where, given an element $x : \alpha$ and a list $\ell : \alpha^{**}$ of lists of elements of type α, *double* appends ℓ to a list derived from adding

[8] In fact, we need to type $\mathbf{D}_t$ so that it can output a Boolean value. Since $\mathbf{D}_t$ is a list used for primitive recursive iteration, its *output type* τ needs to be set to a type Bool coding Boolean values.

x to each list in ℓ. For example, when $\alpha \equiv \mathsf{Bool}$, *double false* $[[\,],[\mathit{true}]]$ reduces to $[[\mathit{false}],[\mathit{false},\mathit{true}],[\,],[\mathit{true}]]$.

$$\begin{aligned}
\mathit{double} &\equiv \lambda x:\alpha.\lambda\ell:(\alpha^*\to\tau\to\tau)\to\tau\to\tau.\\
&\qquad\qquad \lambda c:\alpha^*\to\tau\to\tau.\lambda n:\tau.\\
&\qquad\qquad\qquad \ell\;(\lambda e:\alpha^*.c\;(\lambda c':\alpha\to\tau\to\tau.\lambda n':\tau.c'\,x\,(e\,c'\,n')))\;(\ell\,c\,n)\\
\mathit{double} &: \alpha\to\alpha^{**}\to\alpha^{**}
\end{aligned}$$

Notice that if a λ-term A^* coding a list of λ-terms of type α has type $\alpha^*\equiv(\alpha\to\tau\to\tau)\to\tau\to\tau$ for *any* τ, then A^* also has type $\alpha^*[\tau:=\alpha^{**}]\equiv(\alpha\to\alpha^{**}\to\alpha^{**})\to\alpha^{**}\to\alpha^{**}$. We may then define

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

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:

$$\begin{aligned}
\mathbf{D}_1 &\equiv \mathit{powerset}\;\mathbf{D}_0:\Delta_1\equiv(\Delta_0\to\tau\to\tau)\to\tau\to\tau\\
\mathbf{D}_2 &\equiv \mathit{powerset}\;\mathbf{D}_1:\Delta_2\equiv(\Delta_1\to\tau\to\tau)\to\tau\to\tau\\
&\cdots\\
\mathbf{D}_{k+1} &\equiv \mathit{powerset}\;\mathbf{D}_k:\Delta_{k+1}\equiv(\Delta_k\to\tau\to\tau)\to\tau\to\tau
\end{aligned}$$

In the definition of $\mathbf{D}_{t+1}$, we use $\mathbf{D}_t$ as an iterator, where $\mathbf{D}_t$ is typed as $\Delta_t[\tau:=\Delta_{t+1}]=(\Delta_{t-1}\to\Delta_{t+1}\to\Delta_{t+1})\to\Delta_{t+1}\to\Delta_{t+1}$.

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

Theorem 4.1 *If* $\mathbf{D}_t:\Delta_t$ *as in the above typing,* $|\mathbf{D}_t|=O((b+2t+1)!)$.

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 $|\mathbf{D}_t|=O(n)$. To compute a series of upper bounds $|\mathbf{D}_t|\le d_t$, we define a recurrence where d_{t+1} is computed from d_t. Recall $\mathbf{D}_t\equiv\mathit{powerset}\;\mathbf{D}_{t-1}$, and that to give $\mathbf{D}_t$ the type Δ_t, we take the typing of $\mathbf{D}_{t-1}$ and assign τ the type Δ_t.

If $\mathbf{D}_{t-1}:\Delta_{t-1}[\tau:=\Delta_t]$, then this typing of $\mathbf{D}_{t-1}$ satisfies $|\mathbf{D}_{t-1}|\le d_{t-1}\|\Delta_t\|$. Similarly, if

$$\mathit{powerset}:((\Delta_{t-2}\to\Delta_t\to\Delta_t)\to\Delta_t\to\Delta_t)\to\Delta_t,$$

then as $\|\Delta_{t-2}\|\le\|\Delta_t\|$, we know $|\mathit{powerset}|\le p\|\Delta_t\|$ for a fixed constant p independent of t. We then conclude

$$|\mathbf{D}_t| \;=\; |\mathit{powerset}\;\mathbf{D}_{t-1}| \;\le\; 1+p\|\Delta_t\|+d_{t-1}\|\Delta_t\| \;\le\; (1+p)\|\Delta_t\|+d_{t-1}\|\Delta_t\| \;\le\; 2\|\Delta_t\|d_{t-1}=d_t,$$

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

$$\begin{aligned}
|\mathbf{D}_t| &\le 2\|\Delta_t\|\cdot2\|\Delta_{t-1}\|\cdot\;\cdots\;\cdot2\|\Delta_1\|\cdot d_0\\
&= 2^t d_0\prod_{1\le j\le t}\|\Delta_j\|.
\end{aligned}$$

A simple calculation show that $\|\Delta_j\|=8j+2b+7\le8(b+j+1)$, thus

$$|\mathbf{D}_t| \;\le\; 16^t d_0\prod_{1\le j\le t}(b+j+1) \;=\; 16^t d_0\frac{(b+t+1)!}{(b+t)!},$$

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

$$|\mathbf{D}_t| \quad \leq \quad d_0(b+2t+1)!$$

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

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

4.4 Coding set theory in the $\mathbf{D}_t$

There is a natural idea of *equality* between elements of $\mathcal{D}_t$; when these elements are themselves sets, we can also define the idea of *subset* and of *element* of a set. It is then straighforward 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

$$\begin{aligned} eq_0 \quad \equiv \quad & \lambda x : \aleph . \lambda y : \aleph . \\ & \quad \lambda t : o . \lambda f : o . \\ & \quad\quad x\ (ytff\cdots f)\,(yftf\cdots f)\cdots(yfff\cdots ft) \\ & \quad\quad\quad : \aleph \to \aleph \to \mathsf{Bool} \end{aligned}$$

as a basis, just as we coded $<_0$ over $\mathcal{D}_0$ via tabulation.

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

$$\begin{aligned} member_{t+1} \quad \equiv \quad & \lambda x^t : \Delta_t^{\mathsf{Bool}} . \lambda y^{t+1} : \Delta_{t+1}^{\mathsf{Bool}} . \\ & \quad y^{t+1}\ (\lambda y^t : \Delta_t^{\mathsf{Bool}} . \mathrm{OR}\ (eq_t\ x^t\ y^t))\ false \\ & \quad\quad : \Delta_t^{\mathsf{Bool}} \to \Delta_{t+1}^{\mathsf{Bool}} \to \mathsf{Bool} \\ subset_{t+1} \quad \equiv \quad & \lambda x^{t+1} : \Delta_{t+1}^{\mathsf{Bool}} . \lambda y^{t+1} : \Delta_{t+1}^{\mathsf{Bool}} . \\ & \quad x^{t+1}\ (\lambda x^t : \Delta_t^{\mathsf{Bool}} . \mathrm{AND}\ (member_{t+1}\ x^t\ y^{t+1}))\ true \\ & \quad\quad : \Delta_{t+1}^{\mathsf{Bool}} \to \Delta_{t+1}^{\mathsf{Bool}} \to \mathsf{Bool} \\ eq_{t+1} \quad \equiv \quad & \lambda x^{t+1} : \Delta_{t+1}^{\mathsf{Bool}} . \lambda y^{t+1} : \Delta_{t+1}^{\mathsf{Bool}} . \\ & \quad (\lambda op : \Delta_{t+1}^{\mathsf{Bool}} \to \Delta_{t+1}^{\mathsf{Bool}} \to \mathsf{Bool} . \mathrm{AND}\ (op\ x^{t+1} y^{t+1})\,(op\ y^{t+1} x^{t+1}))\ subset_{t+1} \\ & \quad\quad : \Delta_{t+1}^{\mathsf{Bool}} \to \Delta_{t+1}^{\mathsf{Bool}} \to \mathsf{Bool} \end{aligned}$$

Lemma 4.2 *The λ-terms defining* $member_t$*,* $subset_t$*, and* eq_t *each have length $\Theta(t^2+b^2)$; with type information erased, they have length $\Theta(t+b^2)$.*

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 $\|\Delta_t^{\mathsf{Bool}}\| = \Theta(t+b)$. Notice as well that coding these operations in the typed λ-calculus produces much shorter terms than those that result from coding powerset. □

The above definitions give a typed λ-calculus interpretation to all the logical formulas in our higher-order logic, in the spirit of their standard logical meaning. The elements of $\mathcal{D}_0$ 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 Bool, producing terms of type Bool. Quantifier elimination, as described earlier, interprets $\forall x^t.\Phi(x^t)$ as the iterated conjunction $\mathbf{D}_t\ (\lambda x^t : \Delta_t^{\mathsf{Bool}}.\text{AND}\ (\hat{\Phi}\, x^t))$ *true*, where $\hat{\Phi}$ is the interpretation of Φ; the complementary interpretation of $\exists x^t.\Phi(x^t)$ is the iterated disjunction $\mathbf{D}_t\ (\lambda x^t : \Delta_t^{\mathsf{Bool}}.\text{OR}\ (\hat{\Phi}\, x^t))$ *false*.

When compiling a formula in higher-order logic into a typed λ-term, we can use λ-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 Φ in higher-order logic over the finite base type $\mathcal{D}_0 = \{0, 1, \ldots, b-1\}$ is true if and only if its typed λ-calculus interpretation $\hat{\Phi} : \mathsf{Bool}$ is $\beta\eta$-equivalent to true $\equiv \lambda t : o.\lambda f : o.t : \mathsf{Bool}$. Moreover, if Φ only quantifies over universes $\mathcal{D}_i$ for $i \le t$, then $\hat{\Phi}$ has* order[9] *at most $2t+6$, and if we also write $|\Phi|$ to denote the length of logic formula Φ, then $|\hat{\Phi}| = O(|\Phi|(b+2t+1)!)$.*

4.5 Expressing time bounded computation in higher-order logic

The higher-order logic provides a nice metalanguage to describe typed λ-terms whose intuition is even more inscrutable: the logic lets us talk about the extensional proposition, where the λ-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 compendium of tricks reads like the catalogue of primitive recursive functions in Gödel's first incompleteness theorem. Our goal is to define the formula

$$\exists u.\exists v.\text{INITIAL}(u) \wedge \delta^*(u,v) \wedge \text{FINAL}(v),$$

where $\text{INITIAL}(u)$ expresses that u codes our desired initial configuration, $\text{FINAL}(v)$ expresses that v codes our final, accepting configuration, and $\delta^*(u,v)$ 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 $\mathbf{K}_\ell(n)$ on an input to the Turing Machine of size n. The variables u, v, w must be chosen to range over a universe $\mathcal{D}_k$ 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 ν 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 $\Omega(n \log n)$. We then define the* length $\{\!|\Phi|\!\}$ *of a formula Φ in higher-order logic by structural induction:*

$$\begin{array}{rcl}
\{\!|x^k|\!\} & = & k + \nu(x) \\
\{\!|x^k \in y^{k+1}|\!\} & = & 2k+2 \\
\{\!|x^k = y^k|\!\} & = & 2k+1 \\
\{\!|\neg\Phi|\!\} & = & 1 + \{\!|\Phi|\!\} \\
\{\!|\Phi \circ \Psi|\!\} & = & 1 + \{\!|\Phi|\!\} + \{\!|\Psi|\!\} \\
 & & \circ \in \{\wedge, \vee, \rightarrow\} \\
\{\!|\mathbf{Q}x^k.\Phi|\!\} & = & k + 1 + \{\!|\Phi|\!\} \\
 & & \mathbf{Q} \in \{\forall, \exists\}
\end{array}$$

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

[9] By this bound, we mean that in a type derivation of $\hat{\Phi}$, for any subderivation $\{x_i : \tau_i\} \vdash E : \tau$, the maximum order of any τ_i is $2t+5$, and the maximum order of E is $2t+6$.

4.5.1 Tape 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 $|\mathcal{D}_{t+1}| = 2^{|\mathcal{D}_t|}$, it is easy to show that each element $x^{t+1} \in \mathcal{D}_{t+1}$ 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_1^k and h_2^k code head positions in successive IDs, then $\mathbf{succ}_k(h_1^k, h_2^k)$ expresses that the head moved one position to the right, and $\mathbf{succ}_k(h_2^k, h_1^k)$ expresses that the head moved one position to the left.

We define $\mathbf{succ}_{t+1}$ by first extending the order $<_0$ on $\mathcal{D}_0$ to elements of higher universes:

$$\begin{aligned} x^{t+1} <_{t+1} y^{t+1} \quad \equiv \quad & \exists z^t . z^t \in y^{t+1} \wedge z^t \notin x^{t+1} \\ & \wedge \forall w^t . z^t <_t w^t \to (w^t \in x^{t+1} \leftrightarrow w^t \in y^{t+1}) \end{aligned}$$

(Translation: $x < y$ if the zth 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 $\mathcal{D}_t$ as:

$$\mathbf{succ}_t(x^t, y^t) \quad \equiv \quad x^t <_t y^t \wedge \forall z^t . x^t <_t z^t \to (y^t <_t z^t \vee y^t =_t z^t)$$

Now we fix the cardinality of $\mathcal{D}_0 = \{0, 1, \ldots, b-1\}$ to facilitate the Turing Machine simulation. Since $|\mathcal{D}_t| = \mathbf{K}_t(b)$, we can think of an element of $\mathcal{D}_t$ as coding $\mathbf{K}_{t-1}(b)$ bits (or binary tape cells) of information. We henceforth take $b = \lceil \frac{\log n}{\log \log n} \rceil$; note that for $n \geq 4$, we have $2^{2^b} > n$. As a consequence, $|\mathcal{D}_{\ell+2}| \geq \mathbf{K}_\ell(n)$ for large n, so that an element of $\mathcal{D}_{\ell+3}$ codes $\mathbf{K}_\ell(n)$ 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^t, v^t) simply as the set $\{\{u^t\}, \{u^t, v^t\}\} \in \mathcal{D}_{t+2}$. Because our simulation uses only a small number of pairs, and our universes $\mathcal{D}_t$ 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 λ-term.

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

$$\begin{aligned} \mathbf{zero}_t(s^t) \quad & \equiv \quad \forall b^{t-1} . b^{t-1} \notin s^t \\ \mathbf{con}_t^1(s^t) \quad & \equiv \quad \exists z^t . \mathbf{zero}_t(z^t) \wedge \mathbf{succ}_t(z^t, s^t) \\ \mathbf{con}_t^{m+1}(s^t) \quad & \equiv \quad \exists z^t . \mathbf{con}_t^m(c^t) \wedge \mathbf{succ}_t(c^t, s^t) \end{aligned}$$

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 $\sum b_i 2^i$ to that denoting $\sum b_i 4^i$ by inserting a 0 in between each of the bits b_i:

$$\begin{aligned} \mathbf{shift}_t(x^t, y^t) \quad & \equiv \quad \exists z^{t-1} . \mathbf{zero}_{t-1}(z^{t-1}) \wedge z^{t-1} \notin y^t \wedge \\ & \qquad \forall a^{t-1} . \forall b^{t-1} . \mathbf{succ}_{t-1}(a^{t-1}, b^{t-1}) \to (a^{t-1} \in x^t \leftrightarrow b^{t-1} \in y^t) \\ \mathbf{union}_t(x^t, y^t, z^t) \quad & \equiv \quad \forall w^{t-1} . w^{t-1} \in z^t \leftrightarrow (w^{t-1} \in x^t \vee w^{t-1} \in y^t) \\ \mathbf{spread}_t(x^t, y^t) \quad & \equiv \quad \forall z^{t-1} . z^{t-1} \in x^t \leftrightarrow (\exists w^{t-1} . \mathbf{shift}_{t-1}(z^{t-1}, w^{t-1}) \wedge w^{t-1} \in y^t) \end{aligned}$$

Now we can define a pairing function as

$$\mathbf{pair}_t(a^t, b^t, p^t) \quad \equiv \quad \exists u^t . \exists v^t . \exists w^t . \mathbf{spread}_t(a^t, u^t) \wedge \mathbf{spread}_t(b^t, v^t) \wedge \mathbf{shift}_t(v^t, w^t) \wedge \mathbf{union}(u^t, w^t, p^t)$$

In the last definition, a and b are paired to form p, all in the same universe $\mathcal{D}_t$. Given that an element of this universe only has $\mathbf{K}_{t-1}(b)$ bits, we can only *iterate pairing*[10] a finite number of times, otherwise we run out of "logical memory." Recall, however, the definition $b = \lceil \frac{\log n}{\log\log n} \rceil$; it is not hard to show that for any fixed integer $c > 0$, for sufficiently large n (depending on c), $\mathbf{K}_{j+2}(b) \geq c\mathbf{K}_j(n)$. This essentially means that our logical formula specifying the behavior of a Turing Machine can contain any *fixed* number of pairs, and asymptotically, the "word length" in $\mathcal{D}_{\ell+3}$ is big enough to support that pairing.

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

$$\begin{aligned} \mathbf{Q}(a^t, b^t).\Phi(a^t, b^t) &\equiv \mathbf{Q}p^t.\forall a^t.\forall b^t.\mathbf{pair}_t(a^t, b^t, p^t) \to \Phi(a^t, b^t), \\ (a^t, b^t) \in s^{t+1} &\equiv \exists p^t.\mathbf{pair}_t(a^t, b^t, p^t) \wedge p^t \in s^{t+1}. \end{aligned}$$

4.5.3 Simulating a Turing machine

A small but suitably large $\mathcal{D}_{t+1}$ can code the finite states of the Turing Machine as an integer. To code the tape, recall that an element of $\mathcal{D}_{\ell+3}$ is big enough to code $\mathbf{K}_\ell(n)$ tape cells, so let $k = \ell + 2$; 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:

$$\begin{aligned} \mathbf{ID}(u^{k+1}, h^k, s^k, t^k, z^{k+1}) \equiv\ & \exists v^k.\exists w^k.\exists w^{k+1}. \\ & \mathbf{pair}(h^k, s^k, v^k) \wedge \mathbf{pair}(v^k, t^k, w^k) \wedge \mathbf{single}_k(w^k, w^{k+1}) \wedge \mathbf{pair}(w^{k+1}, u^{k+1}, z^{k+1}) \end{aligned}$$

Note that $\mathbf{single}_k$ is defined as the injection of $a^k \in \mathcal{D}_k$ into the singleton $\{a^k\} \in \mathcal{D}_{k+1}$:

$$\mathbf{single}_k(a^k, a^{k+1}) \equiv a^k \in a^{k+1} \wedge \forall x^k.x^k \in a^{k+1} \to x^k = a^k$$

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 $\mathbf{tape}_i$ 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:

$$\begin{aligned} \textsc{initial}(z^{k+1}) \equiv\ & \exists u^{k+1}.\exists h^k.\exists s^k.\exists t^k \\ & \mathbf{ID}(u^{k+1}, h^k, s^k, t^k, z^{k+1}) \wedge \mathbf{zero}_k(h^k) \wedge \mathbf{zero}_k(s^k) \wedge \mathbf{zero}_k(t^k) \wedge \\ & \quad \exists c_0^k.\mathbf{zero}_k(c_0^k) \wedge \mathbf{tape}_0 \wedge \exists c_1^k.\mathbf{succ}_k(c_0^k, c_1^k) \wedge \mathbf{tape}_1 \wedge \exists c_2^k.\mathbf{succ}_k(c_1^k, c_2^k) \wedge \mathbf{tape}_2 \wedge \cdots \\ & \quad \exists c_{n-1}^k.\mathbf{succ}_k(c_{n-2}^k, c_{n-1}^k) \wedge \mathbf{tape}_{n-1} \wedge \\ & \quad \forall b^k.c_{n-1}^k <_k b^k \to b^k \notin u^{k+1} \end{aligned}$$

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 specificiation 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 $\mathbf{K}_\ell(n)$ steps, we need to be able to specify that number via a succinct formula: we define the time bound as a predicate $\mathbf{T}_j(u^{j+2})$, meaning that u^{j+2} codes $\mathbf{K}_j(n)$:

$$\begin{aligned} \mathbf{T}_0(t^2) &\equiv \mathbf{con}_k^n(t^2) \\ \mathbf{T}_j(t^{j+2}) &\equiv \forall b^{j+1}.b^{j+1} \in t^{j+2} \leftrightarrow \mathbf{T}_{j-1}(b^{j+1}) \end{aligned}$$

If HALT is the constant naming the halting state, the relation defining an accepting configuration is then:

$$\textsc{final}(z^{k+1}) \equiv \exists u^{k+1}.\exists h^k.\exists s^k.\exists t^k.\mathbf{ID}(u^{k+1}, h^k, s^k, t^k, z^{k+1}) \wedge \mathbf{con}_k^{\textsc{halt}}(s^k) \wedge \mathbf{T}_\ell(t^k)$$

Only $\textsc{initial}(z^{k+1})$ and $\textsc{final}(z^{k+1})$ 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 $O(1)$.

[10] By this, we mean expressions like

$$\mathbf{pair}(a_1, a_2, p_1) \wedge \mathbf{pair}(p_1, a_3, p_2) \wedge \mathbf{pair}(p_2, a_4, p_3) \wedge \cdots \wedge \mathbf{pair}(p_{i-1}, a_{i+1}, p_i)$$

Note that if each of the a_j have m bits of information, then p_i needs $j2^i$ bits.

The definition of the transition relation δ is equally straightforward:

$$\begin{aligned}\delta(z_1^{k+1}, z_2^{k+1}) \quad \equiv \quad & \exists u_1^{k+1}.\exists h_1^k.\exists s_1^k.\exists t_1^k. \\ & \quad \exists u_2^{k+1}.\exists h_2^k.\exists s_2^k.\exists t_2^k. \\ & \qquad \mathbf{ID}(u_1^{k+1}, h_1^k, s_1^k, t_1^k, z_1^{k+1}) \wedge \mathbf{ID}(u_2^{k+1}, h_2^k, s_2^k, t_2^k, z_2^{k+1}) \wedge \\ & \qquad \mathbf{succ}_k(t_1^k, t_2^k) \wedge \Phi_{\mathrm{TM}}(u_1^{k+1}, h_1^k, s_1^k, u_2^{k+1}, h_2^k, s_2^k).\end{aligned}$$

The variables z_1^{k+1} and z_2^{k+1} code successive IDs of the Turing Machine; Φ_{TM} codes their relationship as defined by the transition rules of the machine.

It remains only to define the reflexive, transitive closure of δ:

$$\begin{aligned}\delta^*(u^{k+1}, v^{k+1}) \quad \equiv \quad & \exists c^{k+2}. \\ & \quad \big[\forall (r^{k+1}, s^{k+1}). \\ & \qquad \big[(r^{k+1}, s^{k+1}) \in c^{k+2} \leftrightarrow \\ & \qquad\quad r^{k+1} = s^{k+1} \vee \exists q^{k+1}.(r^{k+1}, q^{k+1}) \in c^{k+2} \wedge \ \delta(q^{k+1}, s^{k+1})\big]\big] \\ & \quad \wedge (u^{k+1}, v^{k+1}) \in c^{k+2}.\end{aligned}$$

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 δ.

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 $\mathbf{K}_\ell(|x|)$ 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 $i \leq \ell + 4$, and $\{\!|\Phi_x|\!\} = O(|x|)$.*

By considering the coding of this formula in the simply-typed λ-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 $\mathbf{K}_\ell(n)$ steps. Then there exists an explicitly typed, closed λ-term $\hat{\Phi}_x : \mathsf{Bool}$ such that M accepts x if and only if $\hat{\Phi}_x$ reduces to true $\equiv \lambda t : o.\lambda f : o.t : \mathsf{Bool}$. Moreover, the bound variables in $\hat{\Phi}_x$ have order $2\ell + 13$, and $|\hat{\Phi}_x| = O(n)$.*

Proof. From Theorems 4.1 and 5.1, we know the largest $\mathcal{D}_t$ we need is $\mathcal{D}_{\ell+4}$, coded as λ-term $\mathbf{D}_{\ell+4}$ with order $2\ell + 13$ and length $O((b + 2\ell + 9)!)$. But recall ℓ is a constant and $b = \lceil \frac{\log n}{\log\log n} \rceil$; a straightforward calculation shows that for any fixed constant c, $(b+c)! \leq 2^{c+1}n$; observe that $(b+c)! \leq 2^{(b+c)\log(b+c)}$, and

$$(b+c)\log(b+c) \;=\; (b+c)\left(\log b + \log(1 + \frac{c}{b})\right) \;\leq\; b\log b + c\log b + c + o(1)$$

where

$$b\log b + c\log b \;=\; \log n - \frac{\log n \log\log\log n}{\log\log n} + c\log\log n - c\log\log\log n \;\leq\; \log n$$

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 $O(n)$. $\Box$

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 λ-terms E_n :* Bool*, where $|E_n| = O(n)$, 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 $\Omega(\mathbf{K}_\ell(n))$.*

Proof. Let $\hat{\Phi}_x$ be a typed, closed λ-term that reduces to $\mathit{true} \equiv \lambda t : o.\lambda f : o.t$: Bool if and only if an arbitrary, fixed Turing Machine M accepts x in $\mathbf{K}_{\ell+1}(|x|)$ steps, where the coding of $\hat{\Phi}_x$ is given by Corollary 5.2. By that Corollary and the previous Theorem 5.1, we know that the length of $\hat{\Phi}_x$ is $O(|x|)$. By Theorems 3.12 and 3.13, the number of parallel β-steps needed to normalize $\hat{\Phi}_x$ is also $O(|x|)$.

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

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

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 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 *Let $\ell \geq 0$ be any fixed integer. Then there exists a set of λ-terms E_n :* Bool *which normalize in $O(n)$ parallel β-steps, where the number of local graph interaction steps required to normalize E_n, using Lamping's graph reduction algorithm, grows as $\Omega(\mathbf{K}_\ell(n))$.*

In the previous corollary, the number of graph reduction steps counts not only interactions between λ, 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 [LM96, AG98].

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 $\mathbf{K}_{\ell+1}(|x|)$ 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 ℓ as fixed), where the structure of this graph is given by the "blob" in Figure 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 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 $\Omega(\mathbf{K}_\ell(n))$.*

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 this final corollary:

Corollary 5.6 *Let $\ell \geq 0$ be any fixed integer. 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 of E to normal form. Then there exists an infinite set of λ-terms E where $b_E = \Omega(\mathbf{K}_\ell(p_E))$.*

Proof. Let λ-term E be the η-expanded version of $\overline{n}\overline{2}\cdots\overline{2}$ where $\overline{m}$ is the Church numeral for integer m, and $\overline{n}$ is followed by $\ell+3$ copies of $\overline{2}$. Because $|E| = O(n)$ we know that $p_E = O(n)$. On the other hand, E normalizes to $\overline{\mathbf{K}_{\ell+3}(n)}$, and an ordinary β-step can at most square the length of a λ-term, hence $n^{2^{(b_E)}} = \Omega(\mathbf{K}_{\ell+3}(n))$, from which we derive $b_E = \Omega(\mathbf{K}_{\ell+1}(n))$, hence $b_E = \Omega(\mathbf{K}_\ell(p_E))$. $\square$

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 *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

The analysis of the previous section makes it very clear that the parallel β-step is not even *remotely* a unit-cost operation. The workhorse of any optimal reduction engine is not the parallel β-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 *sanctum sanctorum* where parallel β-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 *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. 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 λ-terms. We have resisted stating theorems in this section, instead encouraging the reader towards a more intuitive grasp of the finitary dynamics of the graph reduction rules.

6.1 Superposition

In Figure 17, 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 code *true*, and the "circle" sides of the sharing nodes

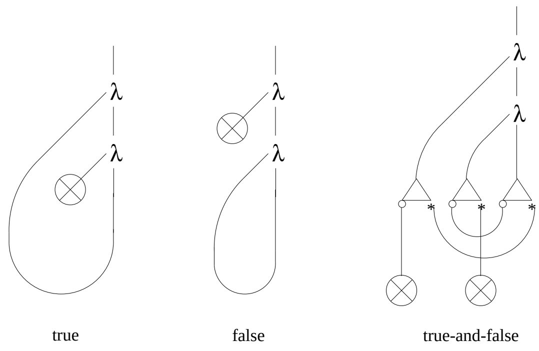


Figure 17: Superposition of true and false.

code *false*. The two values share the λ-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 *one* of them: consider reducing a graph where the function associated with an apply node is NOT, and the associated argument is the superimposed *true* and *false*: the 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 $\mathbf{D}_1 \equiv \lambda c : \mathsf{Bool} \to o \to o.\lambda n : o.c\ \mathit{true}\ (c\ \mathit{false}\ n) : (\mathsf{Bool} \to o \to o) \to o \to o$, and its η-expanded equivalent, pictured in Figure 18. In the unexpanded term, there is a sharing of c, but not a

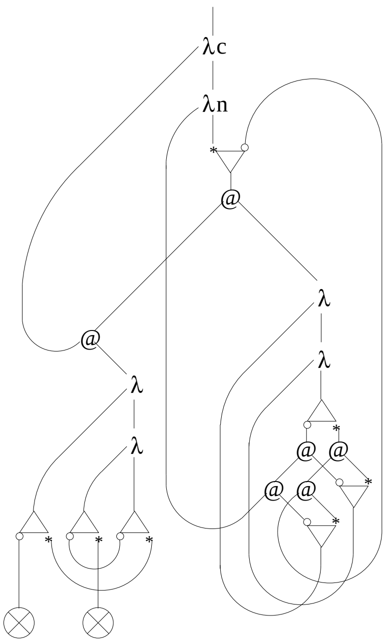


Figure 18: η-expansion of the list of Booleans.

sharing of the four applications. In the η-expanded term, the coding of $c\ \mathit{true}$ and $c\ \mathit{false}$ is shared by a single application, and the argument is exactly our superimposed *true* and *false*. In addition, there is a sharing of $(c\ \mathit{true})\ (c\ \mathit{false}\ n)$ and $(c\ \mathit{false})\ n$. Notice that the companion argument to the superimposed values is just two

η-expansions: the "star" one, coding (c *true*) (c *false* n), leads back into the network of applications (but on the "circle" side), while the "circle" application leads out to the parameter n.

This example explains why SATISFIABILITY can be coded in a sub-exponential number of parallel β-steps. Consider the formula $\exists x.\exists y.\exists z.\phi(x, y, z)$, represented by the following typed λ-term:

$$\begin{array}{l} \mathbf{D}_1\ (\lambda x : \textsf{Bool}.\ \textsc{or} \\ \qquad \mathbf{D}_1\ (\lambda y : \textsf{Bool}.\ \textsc{or} \\ \qquad\qquad \mathbf{D}_1\ (\lambda z : \textsf{Bool}.\ \textsc{or} \\ \qquad\qquad\qquad (\hat{\phi}\ x\, y\, z)\Big)\ \mathit{false}\Big)\ \mathit{false}\Big)\ \mathit{false}. \end{array}$$

Generalizing the above graph constructions, this kind of λ-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 Figure 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

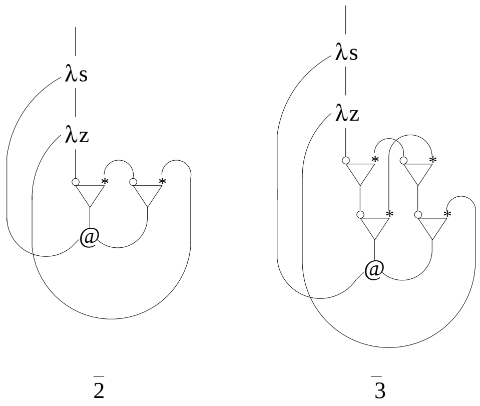


Figure 19: Sharing graph representations of Church numerals.

that these examples give us a fairly universal picture of Church numerals, where the λs and λz serve as a uniform interface, the applications are maximally shared, and two numerals are distinguished only by their network of sharing nodes: see Figure 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 Figure 21.

This style of computation looks like a λ-calculus optimal evalution implementation of SIMD (single instruction, multiple data) parallelism. Imagine a list $\lambda c.\lambda n.c\ x_1\ (c\ x_2\ \cdots\ (c\ x_\ell\ n)\cdots))$ where the x_i are Church numerals, which gets η-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 β-steps, but not counting sharing interactions. 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, factorial) 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. But the actual computation associated with the function occurs via interaction of sharing nodes only, our "parallel computation."

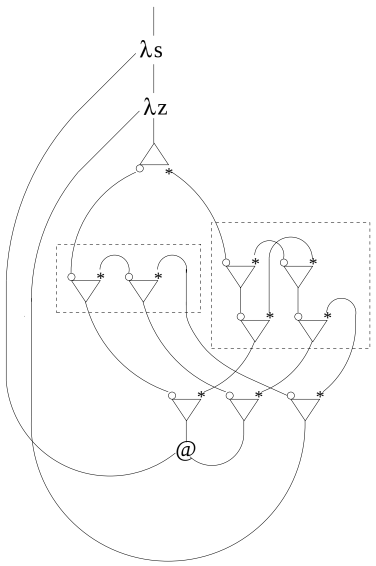


Figure 20: A generic Church numeral.

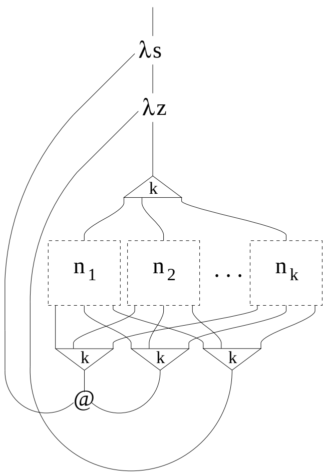


Figure 21: Superimposed Church numerals.

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 β-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 Figure 22.

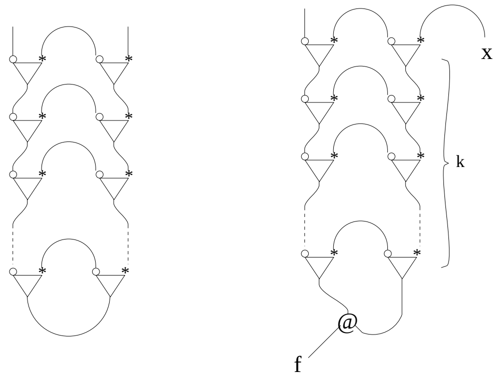


Figure 22: Binary counting and exponential function application.

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.

In the left part of Figure 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 inductive; the basis is described by the simple network of Figure 23, where two binary trees annihilate. By linking two symmetric copies of this construction

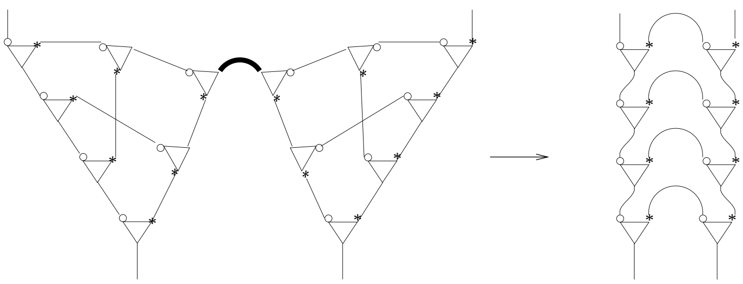


Figure 23: A simple sharing network.

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 Figure 24.

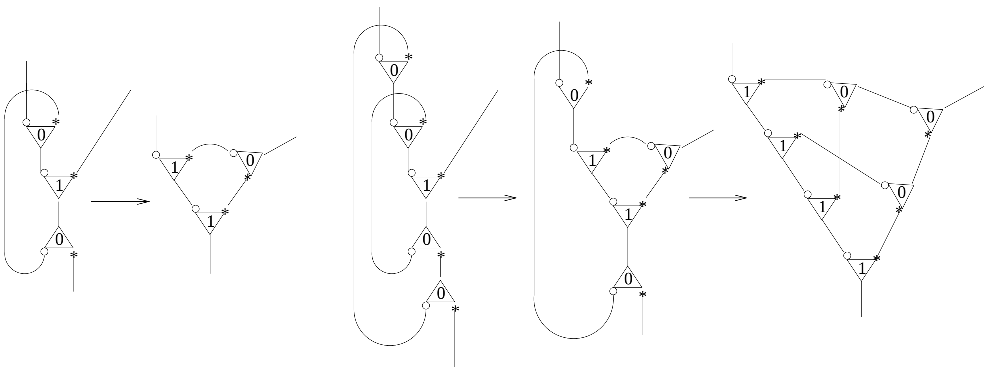


Figure 24: How to share a sharing node.

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^{ℓ} sharing nodes can expand to a network of size $\mathbf{K}_{\ell}(1)$.

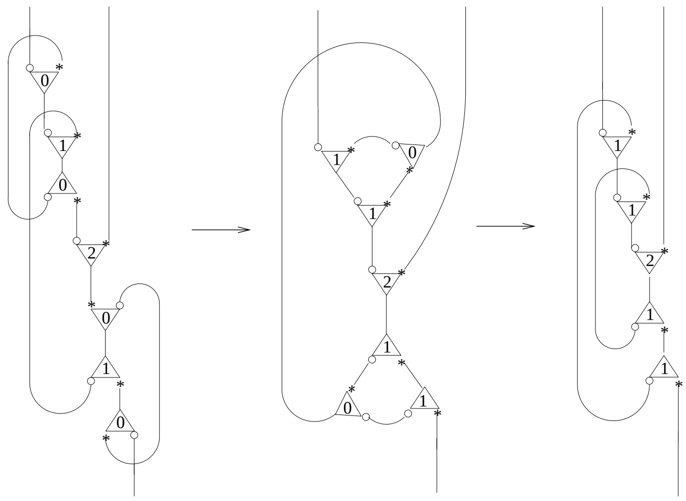


Figure 25: A "nested" construction.

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 $\overline{N}$ 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, $\overline{N_j}$, 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 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.

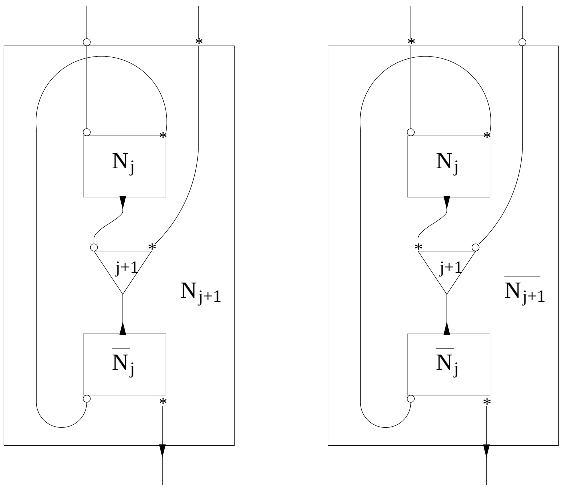


Figure 26: The generic construction of an elementary network.

Lemma 6.1 *The network N_j (respectively, $\overline{N_j}$) normalizes to a complete binary tree with $\mathbf{K}_j(1)$ 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 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 Figure 26. The two binary trees now annihilate each other, and the two stacks of $\mathbf{K}_i(1)$ sharing nodes then create a complete binary tree with $\mathbf{K}_{i+1}(1)$ 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 β-reduction of the η-expanded term $\mathbf{2}_j \equiv \overline{2}\,\overline{2}\cdots\overline{2}$ 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 $\overline{2}$; $\mathbf{2}_j$ normalizes in $O(j)$ parallel β-steps to the Church numeral for $\mathbf{K}_j(1)$. The exponential length is sufficient to code a copy of N_j and $\overline{N_j}$; after normalization, these networks expand to construct $\mathbf{K}_j(1)$ 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 β-reduction steps in the reduction of a λ-term is not bounded in general by any Kalmár-elementary function $\mathbf{K}_\ell(n)$. 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? There is no question that the study of optimal reduction has provided important insights and connections between linear logic, control structures, context semantics and the geometry of interaction, intensional semantics, with hope of 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 β-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 *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 β-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 *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 do learn, beyond the shadow of a doubt, is that merely minimizing the number of parallel β-steps is a curiosity at best, and at 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 β-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 *labelled λ-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 λ-term is initially annotated with a unique label; as reduction occurs, labels are concatenated and underlined, according to the following arcane rule:

$$(\lambda x.E)^{\ell} F \to E^{\underline{\ell}}[x \mapsto F^{\underline{\ell}}]$$

For example, in the untyped λ-calculus, we have the labelled reduction step

$$((\lambda\, x\, .\, (x^1\; x^2)^3)^4\; (\lambda\, x\, .\, (x^5\; x^6)^7)^8)^9 \to\; ((\lambda x.(x^5\, x^6)^7)^{8\underline{4}1} (\lambda x.(x^5\, x^6)^7)^{8\underline{4}2})^{3\underline{4}9}.$$

A labelled λ-term thus codes the reduction history that led to its derivation.

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. This is probably a good thing. It remains, however, to solve the unquestioned algorithmic problem of *spurious bracketing*—the explosion of so-called *bracket* and *croissant* nodes that implement *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 *bracket* and *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 η-expansion method could be generalized to higher-order typed λ-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.

Acknowledgements. 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, Luca Roversi, and René Vestergaard.

References

[Asp94] Andrea Asperti. *Linear logic, comonads, and optimal reductions.* **Fundamenta Informaticae** 22:1, 1994.

[AL95] Andrea Asperti and Cosimo Laneve. *Paths, computations, and labels in the λ-calculus.* **Theoretical Computer Science** 142:2 (May, 1995), pp. 277–297.

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

[AG98] Andrea Asperti and Stefano Guerrini. **The Optimal Implementation of Functional Programming Languages**. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1999.

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

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

[Chu41] Alonzo Church. **The Calculi of Lambda-conversion**. Princeton University Press, 1941.

[Coo71] Stephen A. Cook. *The complexity of theorem-proving procedures.* **3rd Annual ACM Symposium on the Theory of Computing,** pp. 151–158.

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

[Fie90] John Field. *On laziness and optimality in lambda interpreters: tools for specification and analysis.* **1990 ACM Symposium on Principles of Programming Languages**, pp. 1–15.

[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.

[GJ79] Michael R. Garey and David S. Johnson. **Computers and Intractibility: A Guide to the Theory of NP-Completeness.** W. H. Freeman, 1979.

[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.

[HS86] J. Roger Hindley and Jonathan P. Seldin. **Introduction to Combinators and λ-Calculus.** Cambridge University Press, 1986.

[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.

[Kal43] László Kalmár. *Egyszerü példa eldönthetetlen aritmetikai problémára (Ein einfaches Beispiel für ein unentsheidbares 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.

[Laf90] Yves Lafont. *Interaction nets.* **1990 ACM Symposium on Principles of Programming Languages**, pp. 95–108.

[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**, to appear.

[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**, (Jonathan P. Seldin and J. Roger Hindley, editors), 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.

[Sch82] Helmut Schwichtenberg. *Complexity of normalization in the pure typed lambda calculus. The L.E.J. Brouwer Centenary Symposium* (ed. A. S. Troelstra and D. Van Dalen). North-Holland, Amsterdam, 1982.

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

[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.

[Wad71] Christopher Wadsworth. *Semantics and pragmatics of the λ-calculus.* Ph. D. Thesis, Programming Research Group, Oxford University, 1971.