P = NP, up to sharing

Andrea Asperti
Dipartimento di Scienze dell'Informazione
Via di Mura Anteo Zamboni, 7, 40127, Bologna, Italy
asperti@cs.unibo.it

Abstract

We prove that we may compute the normal form of each term of the simply typed λ-calculus in a *polynomial number* of sharable reductions (where the notion of sharing is Lévy's "optimal" one). As a simple corollary, we get that P = NP "up to sharing", i.e. up to the computational overhead due to sharing.

1 Introduction

We prove that we may compute the normal form of each term of the simply typed λ-calculus (up to η-equivalence) in a *polynomial number* of sharable reductions. The notion of sharing is Lévy's one [Le78], commonly known as "optimal" sharing. The general idea (see Section 1.1 for the formal definition) is to formalize duplication of redexes as residuals modulo permutations. In particular, a redex u with history σ (notation σu) is a *copy* of a redex v with history ρ iff $\rho v \leq \sigma u$ (i.e., there exists τ such that $\sigma = \rho\tau$ up to permutation equivalence, and u is a residual of v after τ). The family relation $\simeq$ is then the symmetric and transitive closure of the copy-relation. Two redexes are "sharable" if and only if they belong to a same family.
Lévy's notion obviously depends from the initial term of the derivation. By transforming the input term, we eventually change its families of redexes. This is the clue of our proof. As a matter of fact, a few initial steps of β and particularly η-expansion are enough to guarantee a normalization in a polynomial number of sharable reductions.
The result will be proved using Lamping's algorithm [Lam90] for Levy's optimal reduction. This algorithm, that is used here as a mere tool in our proof, will be introduced in section 3. Having understood the general idea behind Lamping's technique, the proof is completely straightforward (see section 4).
The result is full of consequences. Most notably

1. an alternative proof for the strong normalization of the simply typed λ-calculus (where, conceptually, the "bound" is just polynomial)

2. an amazing (and amusing) relation between sharing and non-determinism: $P = NP$ up to (the cost of) sharing (it simply follows by an encoding of SAT in the simply typed λ-calculus, see section ??).

All these results and their potential developments will be discussed in the final section.

1.1 Sharing

We shall introduce Lévy's notion of sharing in an untyped setting, for simplicity. The whole theory can be rephrased in the typed case with no modification.
Let us consider an example. Let $\Delta = \lambda x.(x\ x)$, $F = \lambda x.(xy)$, $I = \lambda x.x$ and consider the possible reductions of $M = \Delta\ (F\ I)$, represented in Figure 1. Intuitively, there are just three kinds of redexes in this example: R, S and T. In the case of R and S, the common nature of these redexes

is clear, since all R_i and S_i are residuals of R and S. The case of T is more complex. Note that T, T_1 and T_2 are not residuals of any redex in M: they are just created by the immediately previous reduction. The two redexes T_1 and T_2 clearly look sharable (and they are indeed "shared" as the unique redex T in the innermost reduction on the left). However, the only way to establish some formal relation between these redexes is by closing residuals downwards. Indeed, T_1 and T have a common residual T_3 inside the term $(I\ y)(I\ y)$; similarly, T_2 and T have a common residual T_4 in the same term. By transitive closure, we can also conclude the "common nature" of T_1 and T_2: in a sense, T_1 is "the same" redex as T_3, which is the same as T, which in turn is the same as T_4 which is the same as T_2.

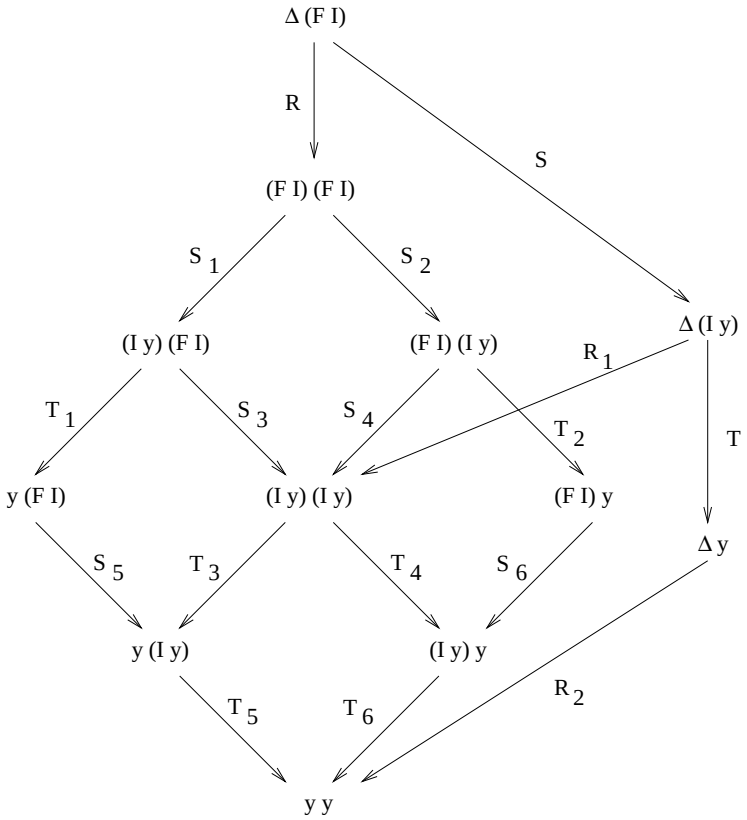

Figure 1: $\Delta = \lambda x.(x\ x), F = \lambda x.(xy), I = \lambda x.x$

So, the idea of "closing residuals downwards", looks enough to connect even redexes which do not have any common ancestor. Our problem, now, is the precise formalization of this intuition.
Note first that T_1 and T_2 should not be related if the initial expression is $(F\ I)(F\ I)$. The intuitive reason is that we are only interested in preserving the sharing "inherent" to the *initial* λ-term, but we are not concerned with the problem of recognising "common subexpressions" created along the reduction (this looks like a mere syntactical coincidence, just like the fact that reducing one or the other redex in $I(I\ x)$ we get the "same" term, and thus should not be taken into account).
A main consequence of the previous assumption is that the relation we are looking for is not really a relation over redexes, but should be relativized to the initial expression. In other words, we should consider each redex R together with the reductions ρ which generated it. In particular, given a derivation ρR, we shall say that ρ is the *history* of R.
Since the redexes we consider are relativized to an initial expression and equipped with an history, they are not simply identified by their syntactical position in the term. Consider for instance the term $M = I(I\ x)$. We have two redexes R and S in M. By firing each of them we obtain the term $N = I\ x$ where we have just a unique redex T. Since T is a residual of both R and S

we could be tempted to conclude that R and S are indeed connected by our "sharing" relation, while these two redexes have clearly nothing to do with each other. The reason of this problem is due to the fact that the redex RT (i.e. T with history R) is definitely different from the redex ST. Another way to look at the same problem is by noticing that the two derivations S and T, although having the same initial and final expressions, are not permutation equivalent, and thus we cannot coherently speak about residuals in N. So, the natural idea is that of taking into account permutation equivalence when closing down the residual relation.

Definition 1.1

copy: *A redex S with history σ is a* copy *of a redex R with history ρ, written $\rho R \leq \sigma S$, if and only if there is a derivation τ such that $\rho\tau$ is permutation equivalent to σ ($\rho\tau \equiv \sigma$) and S is a residual of R w.r.t. τ ($S \in R/\tau$).*

family: *the symmetric and transitive closure of the copy relation is called the* family *relation, and will be denoted with $\simeq$.*

Explicitly, two redexes R and S with respective histories ρ and σ are in a same family ($\rho R \simeq \sigma S$) if and only if $\rho R \leq \sigma S$ or $\sigma S \simeq \rho R$ or there exists some τT such that $\rho R \simeq \tau T \simeq \sigma S$[1].

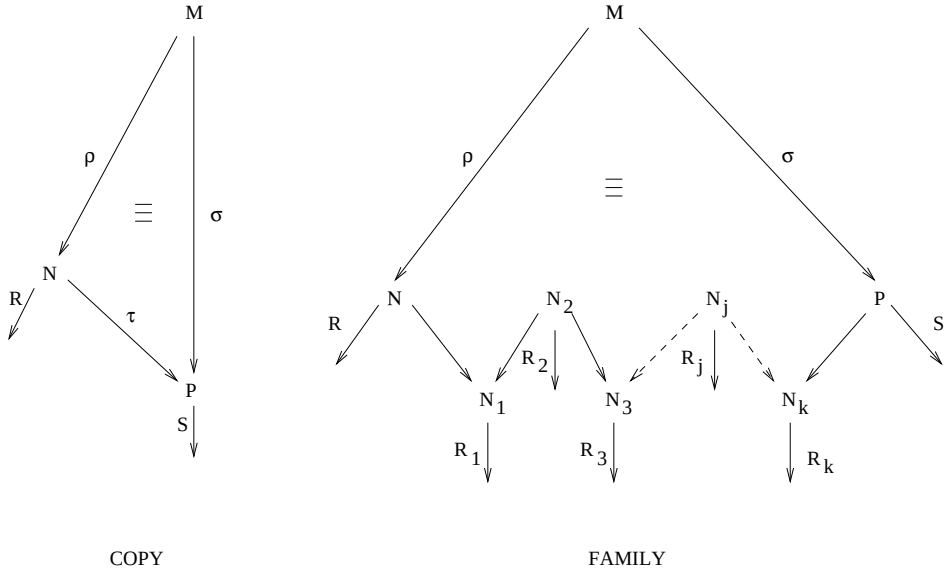

Figure 2: Copy and family relations.

Example 1.2 In the example of Figure ??, one gets $RS_1T_1 \simeq RS_2T_2$. Indeed,

$$RS_1T_1 \leq RS_1S_3T_3 \geq ST \leq RS_2S_4T_4 \geq RS_2T_2$$

□

Definition 1.3 *A* family reduction step *consists in firing all redexes in a same family.*

In this paper, we shall prove that we may always compute the normal form of a simply typed λ-term (up to η-equivalence) in a polynomial number of family reductions.
Note that you should not think of a family reduction step as a sort of "parallel" reduction, but really as a *unique sequential step* where all redexes are (potentially) shared. The actual feasibility

[1] This is also known as the "zig-zag" characterization of the the notion of family, due to the shape of Figure 2.

of Lèvy optimal sharing has been recently proved by Lamping in [Lam90] using a complex graph reduction technique[2] that will be introduced in the next section.
In particular [Lam90, AG97]:

Theorem 1.4 *Lamping's implementation is optimal, in the sense that there is a* bijective correspondence *between β-reduction steps (see next section) and families.*

A crucial point in our proof is that, as we already remarked, Lévy's notion of sharing depends from the inital term of the derivation. As a matter of fact, our first operation is to apply a simple transformation on the initial term t (that consists in a few steps of β and particularly η-expansion), in order to obtain what we call the "optimal root" of t.
In particular, every subterm of the form $\lambda x : \sigma.M$ with multiple occurences of x in M will be transformed in $\lambda x' : \sigma.(\lambda x : \sigma.M\ \eta_\sigma(x'))$, where $\eta_\sigma(x')$ is the (full) η-expansion of x'.
The surprising fact, is that this simple operation is enough to operate such a drastical modification in the "perspective" of the term as to pass from, say, an exponential to a polynomial number of "families".
Our proof is essentially based on Lamping's technique. What we really prove is that, using Lamping's graph reduction algorithm, we may compute the normal form of each term of the simply typed lambda calculus just executing a polynomial number of β-reduction rules.
However, the actual result has very little to do with Lamping's technique: theorem ?? implies that *the number of families* is polynomial!
Lamping's technique is thus used as a mere *tool* for proving the real theorem of the paper. Although it could be surely interesting to look for a more direct proof (without relying on the complex correctness and optimality aspects of Lamping's algorithm), we prefer our approach for several reasons. First of all, as we shall see, the proof becomes almost straightforward. Secondly, this is the very way we got the result, and in our opinion it is thus the best way to present it[3]. This is a good example of a typical research scenario in Computer Science: a nice mathematical theory (Lévy's one) that suggested an impressive algorithm (and a related implementation), that in turns gives important feed-backs on the theory. Let's hope the process will go on.

2 Lamping's graph reduction technique

Lamping's algorithm consists in a complex graph reduction technique for handling sharing in λ-expressions. The crucial point is that we cannot just share expressions, but we must share *contexts*. Moreover, sharing a context, i.e. a term with one or more *holes* inside, requires "unsharing" when exiting through a hole.
Lamping's graph rewriting rules can be naturally classified in two main groups:

1. the rules involving application, abstraction and sharing nodes (fan), that are responsible for β-reduction and duplication (we shall call this group of rules the *abstract system*);
2. some rules involving control nodes (square brackets and croissants), which are merely required for the correct application of the first set of rules.

More precisely, the first set of rules requires an "oracle" to discriminate the correct interaction rule between a pair of fan-nodes; the second set of rules can be seen as an effective implementation of this oracle.
In this paper, Lamping's graph reduction technique is merely used as a convenient tool for proving our result. Since, the "oracle" does not play any role in the proof, we shall not discuss this (complex) part of the algorithm (the interested reader may consult [AG97]).

[2] The actual implementation of Lévy's "optimal" sharing challenged the functional programming community for over fifteen years. Actually, no "traditional" implementation technique, and in particular no technique based on a weak evaluation paradigm (no reduction inside a λ), is optimal in Lévy's sense. The reason is clear: if we consider a term of the kind $\lambda x.M$ as a *value*, we implicitly renounce to the possibility of sharing inside M. This simple fact is illuminating on the actual expressiveness of (sharing in) higher order languages, that gets definetely mortified by that ugly heritage of Algol that is weak evaluation.

[3] We believe that, without Lamping, the result had still to wait for quite a long time

2.1 Initial translation

We shall not bother the reader with the definition of the simply typed λ-calculus. Let us just recall that types $\mathcal{T}$ are generated from an atomic type o by means of the arrow construct $\rightarrow$, i.e. $o \in \mathcal{T}$, and $\alpha \rightarrow \beta \in \mathcal{T}$ provided $\alpha, \beta \in \mathcal{T}$.
Initially, in the optimal graph reduction technique, a λ-term is essentially represented by its abstract syntax tree (like in ordinary graph reduction). There are two main differences, however:

1. we shall introduce an explicit node for sharing;
2. we shall suppose that variables are explicitly connected to their respective binders.

Moreover, since we work in a typed setting, we shall label each edge of the graph with a suitable type (in these graphs, we shall usually use the notation β^α instead of $\alpha \rightarrow \beta$, since it is more compact). Let us remark that types do not play any role in the reduction of the term; they are just an invariant of the computation.
For instance, the graph in Figure 6.(1) is the initial representation of the λ-term $M = (two_{o \rightarrow o}\ two_o)$, where $two_\alpha = \lambda x : \alpha \rightarrow \alpha . \lambda y : \alpha . (x(xy)) : (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$. The type $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$ is the type of Church integers (or iterators) for α, and is usually denoted $\mathcal{N}_\alpha$.
The triangle (we shall call it *fan*) 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). Since multiple occurrences of a same variable are shared by fans, we shall have a single edge leaving a λ towards its variables. So, each node in the graph (@, λ and fan) has exactly three distinguished sites (ports) where it will be connected with other ports. One of this ports (depicted with an arrow in Figure 3) is called *principal*: it is the only port where the node may interact with other nodes (see the interaction

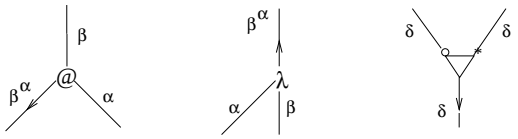

Figure 3: Nodes, ports and their types.

rules, below). The other ports are called *auxiliary*. The graph (and the corresponding term) 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 3 (whose reading should be clear).

Notation We call type of a node the type of its principal port.

Note that, instead of typing edges, we could equivalently type each port of a node (adding, moreover, a suitable polarity). Our approach has been essentially adopted for typographical reasons.

2.2 Reduction

We shall now illustrate the main ideas of Lamping's optimal graph reduction technique by showing how it would work on our sample term $(two_{o \rightarrow o}\ two_o)$. As we shall see, a crucial issue will remain unresolved. This is exactly where the oracle comes in: however, since the understanding of the actual implementation of the oracle is not necessary for our purposes, we shall not discuss this complex topic here.
Lamping's algorithm consists of a set of local graph rewriting rules. At a given stage of the computation, we can usually have several reducible configurations in the graph. In this case, the choice of next rule to apply is made non-deterministically. This does not matter that much, since the graph rewriting system is an Interaction Net in Lafont's sense [Laf90], and it satisfies a one-step diamond property (that implies not only confluence but also that, if a term has a normal

form, all normalizing derivations have the same length). In particular, we shall usually choose the next rule in our example of reduction according to a dydactical criterion (and sometimes for graphical convenience).
The most important of the graph rewriting rules is obviously β-reduction: $(\lambda x.M\ N) \rightarrow M[N/x]$. In graph reduction, substituting a variable x for a term N amounts to explicitly connect the

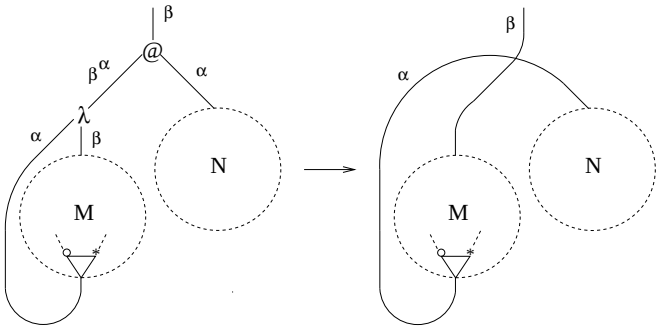

Figure 4: β-reduction

variable to the term N. Moreover, the value returned by the application before the redex is fired (the link above the application) becomes the (instantiated) body M of the function. Since the portions of graph representing M and N do not play any role in the graph reduction corresponding to the β-rule, this reduction can be expressed by the completely local graph rewriting rule in Figure 5.

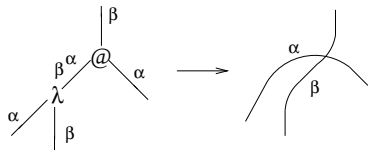

Figure 5: the β-rule

Note that the rule preserves the correct typing of the graph (i.e. it connects edges with equal type). This is a straightforward property of all rewriting rules of Lamping's algorithm, and we shall avoid repeating it each time.
By firing the outermost β-redex in $(two\,\delta)$, we get the graph in Figure 6.(2). Since the next redex involves a shared λ-expression, we must eventually proceede to its duplication. In ordinary graph reduction, this duplication would be performed as a unique, global step on the shared piece of graph. On the contrary, the optimal graph reduction technique proceedes in a more lazy way, duplicating the external λ but still sharing its body. However, since the binder has been duplicated, we are forced to introduce another fan on the edge leading from the binder to the variable.

In a sense, this fan works as an "unsharing" operator (fan-out, usually depicted upside-down), that is to be "paired" against the fan(-in) sharing the body of the function[4]. 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 8. Note that the type fan nodes (the type of their principal port) may only decrease by the firing of this rule.

[4] Although there is no operational distinction between a fan-in and a fan-out, their intutive semantics is quite different; in particular, keep in mind 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 way the correct pairing between fans is determined is a crucial point of the optimal graph reduction technique, solved by the "oracle". Obviously, in order to give a precise definition of the abstract system we should provide the formal definition of the oracle, that is very complex and would just obscure the intuitive nature of the abstract rules. In particular, the obvious and naive idea of labeling fans does not work (see [?]).

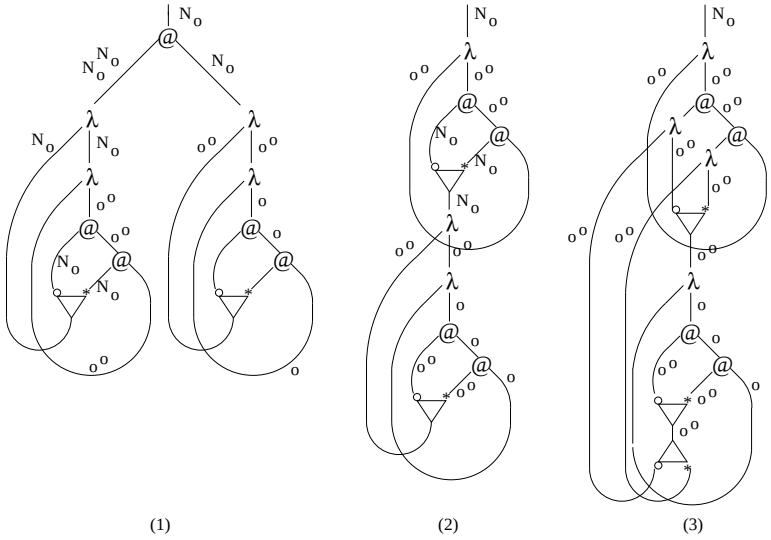

Figure 6: Graph reduction of (*two two*)

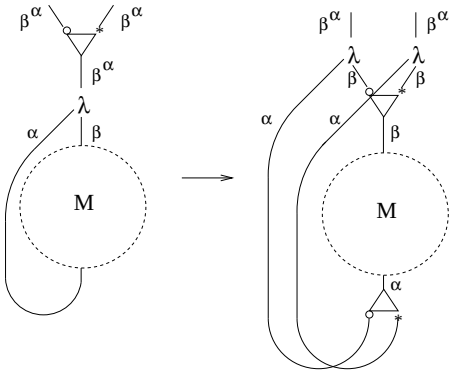

Figure 7: the duplication of abstractions

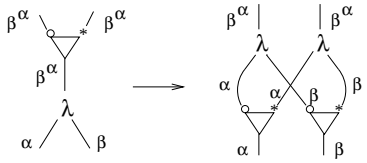

Figure 8: Fan-λ interaction

Let us now proceede in the analysis of our example. By applying the fan-λ interaction rule, we get the graph in Figure 9.(3). Now, two β-redexes have been created, and by firing them we get the graph in Figure 9.(4). Then let us fire the fan-λ rule, obtaining the graph in Figure 9.(5). We have no more β-redexes in this graph, and no fan-λ interactions, so we must proceede in

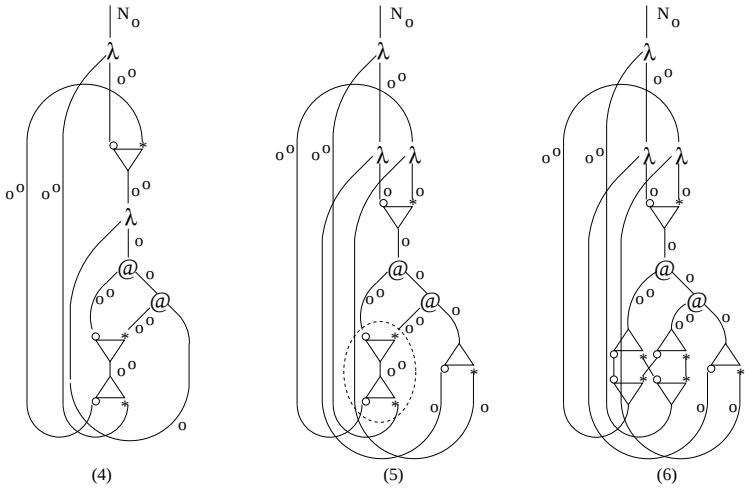

Figure 9: Graph reduction of $(\boldsymbol{two}\ \delta)$

the duplication process, but we must be very carefully here. In particular, the following graph rewriting rule is strictly forbidden, in the optimal imlementation technique (although semantically

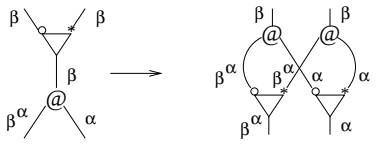

Figure 10: not optimal duplication of the application

correct). The intuition should be clear: since the shared application could be involved in some redex, its duplication would imply a double execution of the redex.
The only other possible interaction is between the two fans inside the dotted region. This is another crucial point of the optimal graph reduction technique. As we shall see, this interaction must be handled in a different way form the similar interactions in Figure 9.(7). Note in particular that the two fans in Figure 9.(5) are not "paired": the fan-in is a residual of the shared variable of $\boldsymbol{two}_o$, while the fan-out is a residual of the shared variable of $\boldsymbol{two}_{o\to o}$, in the process of duplicating $\boldsymbol{two}_o$. Since the two fans have nothing to do with each other, they must duplicate each other, according to the rule in Figure 11.(2). Note that the type of fan-nodes is preserved by this interaction.

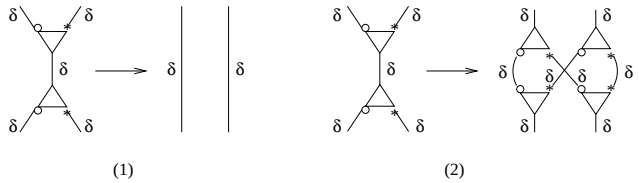

Figure 11: fan-annihilation rule

Now (see Figure 9.(6)), we have a fan-out in front of the function-port of the application. In this case, we can apply the following rule: Intuitively, this rule is correct from the point of view of

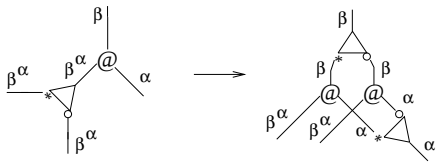

Figure 12: fan-@ interaction

optimal sharing since such a configuration already implies the existence of two unsharable (class of) redexes for the application. As in the case of fan-λ interaction, the type of the fan node strictly decreases.
By firing twice this rule, we get the graph in Figure 13.(8), where we have three pairs of fans

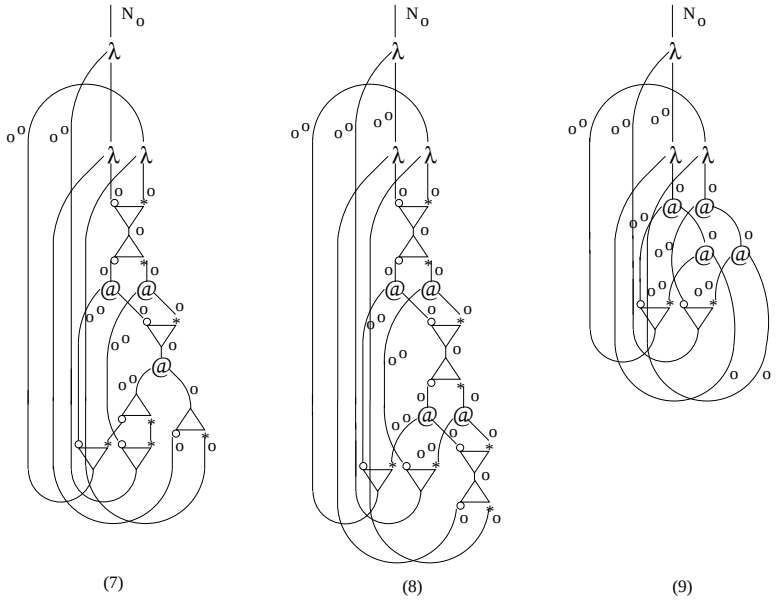

Figure 13: Graph reduction of ($\boldsymbol{two}\ \delta$)

interacting with each other. In this case, all fans are paired: they both belong to the same "duplication process", that has been now (locally) completed. So, the obvious rule, in this case, is to annihilate the paired fans, according to Figure 11.(1).
The problem of deciding which rule to apply when two fans meet each other (that is the question of how their pairing is established) is a difficult point of the optimal implementation technique (solved by the oracle). However, its understanding is not essential for our purposes.
By three applications of the fan-annihilation rule, we get the graph in Figure 13.(9). The three last steps are, respectively, a fan-λ rule, and two β-reductions. The term in Figure 14.(12) is in normal form w.r.t. Lamping's algorithm. We leave as an exercise to the reader to understand why this graph represents (as we expect) the Church integer $\boldsymbol{four}$. The simplest way is to complete the duplication process by applying the "forbidden" rule of duplication of the application; alternatively, you can try to read-back the term by travelling inside its structure: the only proviso to respect, here, is to exit from a fan-out form the same side ($\star$ or $\circ$) we entered the paired fan-in.
Finally, let us remark one more time the major property that will play an essential role in our proof: the type of fan-nodes cannot grow along the reduction.

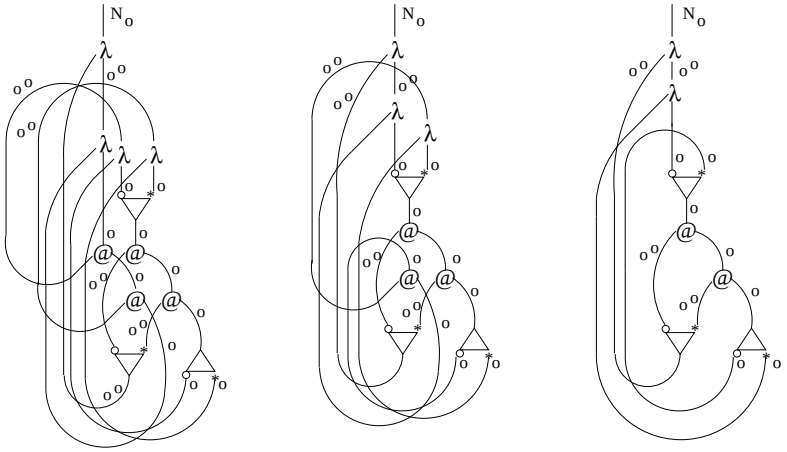

Figure 14: Graph reduction of $(two\ \delta)$

3 The expansion technique

Definition 3.1 *Let $x : \sigma$ be a variable of type σ. The* eta-expansion $\eta_\sigma(x)$ *of x is the term inductively defined on σ in the following way:*

1. $\eta_o(x) = x$
2. $\eta_{\alpha_1 \to \ldots \to \alpha_n \to o}(x) = \lambda y_1 : \alpha_1. \ldots \lambda y_n : \alpha_n.(x\ \eta_{\alpha_1}(y_1)\ \ldots\ \eta_{\alpha_n}(y_n))$

We shall often omit subscript when the type of variables is clear form the context.
For instance, let $x : (o \to o) \to o \to o$. Then, $\eta(x) = \lambda y : o \to o.\lambda z : o.(x\ \lambda w : o.(y\ w)\ z)$.
In graph notation, each eta-expanded variable is a piece of graph[5] with exactly two "entries", that we shall respectively call the positive and negative entries of the variable (see Figure: 15). The

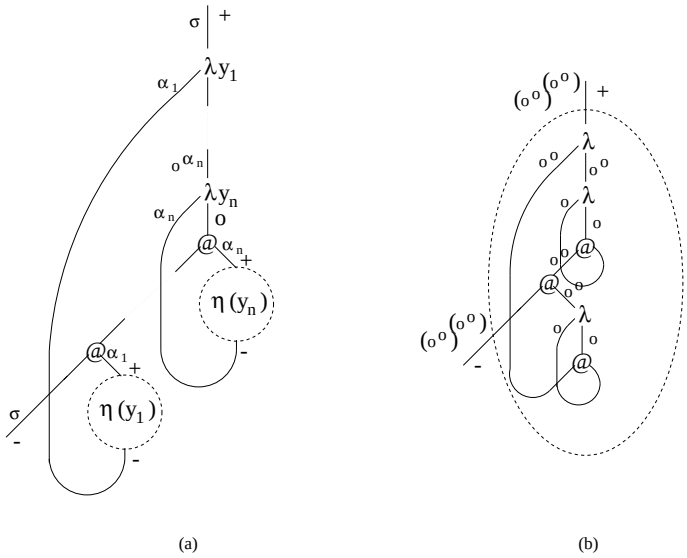

Figure 15: (a) η-expansion; (b) $\eta_{(o\to o)\to o\to o}(x)$.

[5] If the variable has atomic type, this piece of graph is just an edge.

interesting property of these graphs from the point of view of optimality is that any attempt of duplicating them will end up in a configurationb where *all fans have atomic types.*

Proposition 3.2 *Suppose to apply a fan at one of the two entries*[6] *the eta-expansion* $\eta(x)$ *of a variable* x*, and let* $\Delta(x)$[7] *be its normal form w.r.t. Lamping's algorithm. Then, all residual fans in* $\Delta(x)$ *have atomic types.*

*Proof.*By induction on the type σ of the variable x.
If x has atomic type, we have nothing to prove: the fan is $\Delta(x)$.
Suppose $\sigma = \alpha_1 \to \ldots \to \alpha_n \to o$. Then, $\eta_\sigma(x)$ has the shape in Figure 15(a). If we apply a fan at the positive entry, it will reduce to the graph in Figure 16(1); respectively, if we apply a fan at the negative entry, it will reduce to the graph in Figure 16(2). In both cases, the result follows by

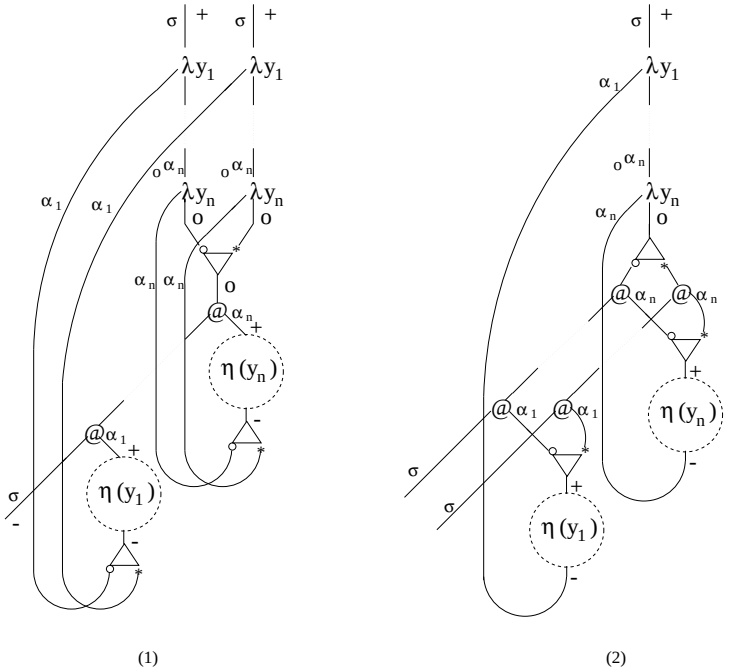

Figure 16: Fan propagation inside $\eta(x)$.

induction. □

Example 3.3 The Graph in Figure 17 is $\Delta(\eta_{(o\to o)\to o\to o}(x))$. □

Obviously, the previous result may be trivially generalised to an arbitrary sequence of fan (even better, you should think in terms of n-ary fans, or *multyplexers*, in Guerrini's terminology).

Definition 3.4 *Let* M *be a simply-typed* λ*-term. The* optimal root $or(M)$ *of* M *is the term obtained by applying the following transformation: for any subterm of* M *of the form* $\lambda x : \sigma.P$ *where* $\sigma \neq o$ *and* x *is (syntactically) shared in* P*, replace* $\lambda x : \sigma.P$ *with* $\lambda x'.(\lambda x.P\ \eta_\sigma(x'))$*. The new* β*-redexes introduced by this transformation will be called* preliminar *redexes.*

Clearly, the transformation is applied just once for any subterm of the form $\lambda x : \sigma.P$. Note that $or(M)$ is obtained by M by means of η and β-expansions. So, $M =_{\beta\eta} or(M)$.
The previous transformation should be correctly appreciated: the idea is that instead of duplicating the actual parameter which will replace the variable x after the firing of its binder, we just duplicate, once and for all, the "skeleton" of this term, provided by the η-expansion of the variable.

[6] As it will become clear, we are mainly interested to the case the fan is applied to the positive entry of $\eta(x)$; however, in the proof we need the more general statement of proposition of 3.2.

[7] $\Delta(x)$ stands for "duplicated" x.

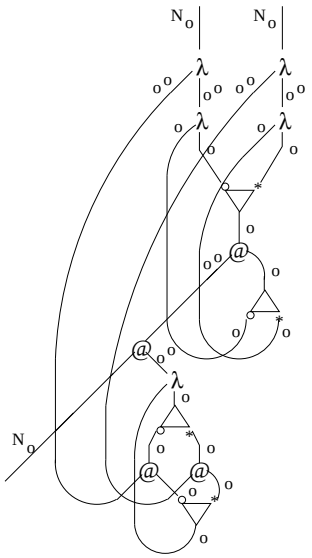

Figure 17: Duplication of $\eta_{(o\to o)\to o\to o}(x)$.

Definition 3.5 *$\Delta(M)$ is the graph in normal form w.r.t. Lamping's algorithm obtained by $or(M)$ by firing all prelimanr redexes (and no other redex).*

The crucial property of $\Delta(M)$ is the following:

Theorem 3.6 *All fans in $\Delta(M)$ have atomic types.*

Proof. Note that, after firing the preliminar redexes, all fans are "on top" of η-expanded variables. So, after (structural) normalization, that amounts to broadcasting fans inside the η-expanded variables, all fans will have atomic types by Proposition 3.2. □

Essentially, $\Delta(M)$ is obtained from M by replacing the portion of graph representing a shared variable x with $\Delta(x)$.
As an obvious consequence of Theorem 3.6, we have:

Theorem 3.7 *The number of β-reductions (and thus of families) fired along the normalization of $\Delta(M)$ cannot exceed its initial size.*

Proof. Since all fans have atomic types in $\Delta(M)$ they will never possibly interact with abstractions or applications. As a consequence, no new application or λ-node will be created along the reduction, and the number of β-redexes in Lamping's algorithm is thus bounded by the intial size of $\Delta(M)$. □

We have still to prove the the size of $\Delta(M)$ does not grow too much with respect to M.
Let us suppose to rename all variables in M in such a way that different abstractions refers to different variables (so, we shall refer to each variable by its name with no ambiguity). Let $n(x)$ be the multiplicity of x in M. Given a type σ let $|\sigma|$ its structural size (defined in the obvious way). Note that the size of $\eta_\sigma(x)$ is linear in $|\sigma|$; in particular, $|\eta_\sigma(x)| \leq 2 * |\sigma|$. Moreover, $|\Delta_{n(x)}(x)| \leq (n(x) * |\eta_\sigma(x)|) \leq 2 * n(x) * |\sigma(x)|$. Considering that the initial β-redexes are bounded by $|M|$, we have:

Theorem 3.8 *The number f of families of $or(M)$ is less than*

$$2 * (|M| + \sum_{x \in M} n(x) * |\sigma(x)|)$$

We can further simplify this formula. Let t be the maximal size among the type of the variables in M. Then

$$f \leq 2 * (|M| + t * \sum_{x \in M} n(x))$$

But, obviously, $\sum_{x \in M} n(x) \leq |M|$. So

$$f \leq 2 * |M| * (t + 1)$$

In particular, if the type of the term is fixed (that is the common case), the number of families in $or(M)$ is just *linear* in the size of M.
Another interesting consequence of Theorem 3.6 is the following:

Proposition 3.9 *In $\Delta(M)$, all β-rules can be fired before any other interaction.*

Proof. Just observe that if all fans have atomic types, no fan-interaction rule may create a β-interaction. So, new β-interactions are just possibly created by firing previous β-interactions, till exhaustion. □

If you think of $\Delta(M)$ as the representation of a logical proof via the Curry-Howard analogy (warning: you have to make sense of "fan-out"), it gives a very interesting "normal" form, where all logical rules are "below" the structural ones. This is amusing: you can immediately get rid of logical cuts. The rest of the computation is merely structural, i.e. annihilations or duplications of fans.
A major consequence of the finite number of families in $or(M)$ is its Strong Normalization (and, all the more reason, the Strong Normalization of M). This is a trivial corollary of the following result of Lévy ([Le78], Theorem 4.4.6., p.65):

Theorem 3.10 *Let σ be any finite (possibly parallel) reduction of a term M. Then any reduction σ'* relative[8] *to σ is terminating.*

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

Theorem 3.11 *The simply tuped λ-calculus is Stongly Normalizing.*

Even more, the "bound" of the reduction is, conceptually, just polynomial (i.e. polynomial w.r.t. families).

4 The Satisfability Problem

In this section we prove that the Satisfability Problem of Propositional Logic (SAT) can be encoded in the simply typed λ-calculus. Types are not uniform, but they just grow linearly in the number of free variables of the input formula. As a simple corollary, we shall obtain that P = NP up to (the cost of) sharing.

In the following, we shall use the notation $\alpha^n \rightarrow \beta$ for the type $\alpha \rightarrow \ldots \rightarrow \alpha \rightarrow \beta$ with n occurrences of α.
Let us start recalling the usual encoding of booelans.

Definition 4.1 *Let $Bool_\sigma = \sigma \rightarrow \sigma \rightarrow \sigma$ ($Bool_o$ will be abrreviated in $Bool$, in the following). Then:*

$T_\sigma : Bool_\sigma = \lambda x : \sigma.\lambda y : \sigma.x$
$F_\sigma : Bool_\sigma = \lambda x : \sigma.\lambda y : \sigma.y$
$neg_\sigma : Bool_\sigma \rightarrow Bool_\sigma = \lambda b : Bool_\sigma.\lambda x : \sigma.\lambda y : \sigma.(b\ y\ x)$
$and_\sigma : Bool_\sigma \rightarrow Bool_\sigma \rightarrow Bool_\sigma = \lambda b1 : Bool_\sigma.\lambda b2 : Bool_\sigma.\lambda x : \sigma.\lambda y : \sigma.(b1\ (b2\ x\ y)\ y)$
$or_\sigma : Bool_\sigma \rightarrow Bool_\sigma \rightarrow Bool_\sigma = \lambda b1 : Bool_\sigma.\lambda b2 : Bool_\sigma.\lambda x : \sigma.\lambda y : \sigma.(b1\ x\ (b2\ x\ y))$

[8] σ' is relative to σ if all redexes in σ' are in the same family of same redex in σ, see [Le78], Def. 4.4.1., p.64).

By composition, each propositional formula $\mathcal{P}$ with atomic proposition in $x_1 \ldots x_n$ can be easily encoded with a term P_σ of type $Bool^n_\sigma \to Bool_\sigma$ (we shall fix σ later on).
The general idea for establishing the satisfability of P is very simpe. We shall work with a tuple of $n+1$ boolean values $\langle b_0, b_1, \ldots, b_n\rangle$. The idea is that b_0 will hold the current value computed for P, while $b_1, \ldots, b_n$ define the current assignament for $x_1 \ldots x_n$. Then we iterate the following function:

$$\lambda\langle b_0, b_1, \ldots, b_n\rangle.\langle b_0 \wedge (P b_1 \ldots b_n), b'_1, \ldots, b'_n\rangle$$

where $b'_1, \ldots, b'_n$is the next assignement (i.e. the "successor" of $b_1, \ldots, b_n$).
This function must be iterated exactly 2^n times; then we shall keep the first projection of the tuple, that is our result.
The only tricky point is the definition of the "successor".
The first element b_0 of our tuple will have type $Bool$; the other elements, instead, will have type $Bool_{Bool}$.
Before entering in the encoding, let us define a few important types that we shall use in the following:

$$\rho_n = Bool^n_{Bool} \to Bool$$

$$\pi_n = Bool \to \rho_n$$

$$\tau_n = \pi_n \to Bool$$

The tuple $\langle b_0, b_1, \ldots b_n\rangle$ will be represented by a lambda term of the kind:

$$\lambda z : \pi_n.(z\ b_0\ b_1\ \ldots\ b_n) : \tau_n$$

Let us define the following functions (where $F = F_{Bool}$ and $T = T_{Bool}$):

$s_1 = \lambda w : \rho_1.\lambda b_1 : Bool_{Bool}.(b1\ (w\ F)\ (w\ T)) : \rho_1 \to \rho_1$
$s_2 = \lambda w : \rho_2.\lambda b_1 : Bool_{Bool}.\lambda b_2 : Bool_{Bool}.(b_1\ (b_2(w\ F\ F)(w\ F\ T))\ (w\ T\ b_2)) : \rho_2 \to \rho_2$
$s_3 = \lambda w : \rho_3.\lambda b_1 : Bool_{Bool}.\lambda b_2 : Bool_{Bool}.\lambda b_3 : Bool_{Bool}.$
$\quad (b_1(b_2(b_3(w\ F\ F\ F)\ (w\ F\ F\ T))(w\ F\ T\ b_3))\ (w\ T\ b_2\ b_3)) : \rho_3 \to \rho_3$
$\ldots$
$s_n = \lambda w : \rho_n.\lambda b1 : Bool_{Bool}.\lambda b_2 : Bool_{Bool}.\lambda b_3 : Bool_{Bool}.\ldots.\lambda b_n : Bool_{Bool}.$
$\quad (b_1(b_2(\ldots(b_n(w\ F \ldots F\ F)\ (w\ F \ldots F\ T))\ldots)(w\ F\ T; b_2 \ldots b_n))\ (w\ T\ b_2\ b_3 \ldots b_n)) : \rho_n \to \rho_n$

Given a propositional formula $P : Bool_{o\to o} \to \ldots \to Bool_{o\to o} \to Bool_{o\to o}$ let us define

$$P_{flat} = \lambda b_1 : Bool_{Bool}.\ldots.\lambda b_n : Bool_{Bool}.(P\ b_1 \ldots b_n\ T_o\ F_o) : \rho_n$$

We are now ready to define the iteration function $next_n$:

$$next_n = \lambda t : \tau.\lambda z : \pi.(t\ (\lambda b_0 : Bool.(s_n\ (z\ (or_o\ b_0\ (t\ \lambda b_0 : Bool.P_{flat})))))) : \tau_n \to \tau_n$$

The starting tuple of the iteration is:

$$start_n = \lambda z : \pi.(z\ F_o\ F_{Bool} \ldots F_{Bool}) : \tau$$

The final projection is

$$fst_n = \lambda b_0 : Bool.\lambda b_1 : Bool_{Bool}.\ldots.\lambda b_n : Bool_{Bool}.b_0 : \pi_n$$

Finally, we must define the iterator 2^n. Let $two_\sigma = \lambda x : \sigma \to \sigma.\lambda y : \sigma.(x(x\ y)) : \mathcal{N}_\sigma$. Then,

$$two^n = \lambda f : \sigma \to \sigma.(two_\sigma \ldots (two_\sigma(two_\sigma\ f))\ldots) : \mathcal{N}_\sigma$$

with n applications of two_σ. In particular, we are interested in two^n_τ.
The satifability of P can thus be tested by the term

$$sat_n = (two_\tau^n\ next_n\ start_n\ fst_n) : Bool$$

The term sat_n grows linearly in the size of $\mathcal{P}$, and quadratically in n (due to s_n). Its types are constant in the size of $\mathcal{P}$ and linear in n. So, by a rough application of our result, we discover that the (family) reduction of $or(sat_n)$ is linear in the size of $\mathcal{P}$ and cubic in n.
However, a tidier inspection of the term shows that it is merely *quadratic* in n. Indeed s_n is the only subterm that grows quadratically. However, the number of occurrences of w grows linearly, as well as its type. The sum of the number of occurrences of the other variables grows quadratically, but in this case their type is constant (it is $Bool_{Bool}$).
As an immediate corollary, we get:

Theorem 4.2 *$P = NP$, up to (the cost of) sharing.*

The previous result has also been experimentally confirmed by our prototype implementation of (a variant of) Lamping's algorithm: the Bologna Optimal Higher-Order Machine (BOHM)[9]. BOHM [AG97] is a simple interpreter written in C. Its source language is a sugared λ-calculus enriched with booleans, integers, lists and basic operations on these data types. The extension of Lamping's technique to this language is essentially based on Asperti and Laneve's work on Interaction Systems [?]. In particular, all syntactical operators are represented as nodes in a graph. These nodes are divided into *constructors* and *destructors*, and reduction is expressed as a local interaction (graph rewriting) between constructor-destructor pairs.
BOHM is *lazy* (in the sense that it always reduces the *family* of the leftmost outermost redex) and *weak* (that is, it stops computing as soon as the top node in the graph is a constructor). As a consequence, BOHM supports lazy data structures, such as streams.
Due to its prototyping nature, BOHM has been especially designed to provide a large number of experimental data relative to each computation (user and system time, total number of different kinds of interactions, storage allocation, garbage collection, and so on).
As an example, in Figure ?? we give the results of the satisfability test for propositional formulas P_n defined as conjunction of n distinct atomic symbols. The first column refers to $sat_n(P_n)$, while the second column refers to its optimal root.

	$sat_n(P_n)$			$or(sat_n(P_n))$		
n	β-red.	tot. inter.	user	β-red.	tot. inter.	user
1	47	251	0.00	85	374	0.00
2	85	910	0.00	152	1453	0.01
3	135	3040	0.03	239	4549	0.05
4	201	10752	0.10	346	15206	0.13
5	291	40315	0.35	473	55113	0.45
6	421	156629	1.42	620	208549	2.19

Figure 18: The function g

The three columns for each term are, respectively, the number of β-redexes (family reductions) fired along the computation, the total number of interactions (comprising the "oracle") and the user time (the test has been performed on a Sparc-station 5).
In particluar, the number of β-reductions grows as

$$2 * 2^n + 4 * n^2 + 22 * n + 17$$

[9]BOHM is available by anonymous ftp at ftp.cs.unibo.it, in the directory /pub/asperti. Get the file BOHM1.1.tar.Z (compressed tar format).

for $\boldsymbol{sat}_n(P_n)$, while it is just quadratic for $\boldsymbol{or}(\boldsymbol{sat}_n(P_n))$:

$$10 * n^2 + 37 * n + 38$$

5 Conclusions and future work

In this paper, we proved that every term of the simply typed λ-calculus can be normalized (up to η equivalence) in a *poynomial* number of family reductions. This result reveals the genuinely polynomial nature of this formalism. Many efforts have been recently devoted to the problem of finding "intrinsically polynomial logical systems". Well, here it is[10]: the good old simply typed λ-calculus.
The natural question is then: what about System F? In particular, an important corollary of our result is the Strong Normalization of the simply typed λ-calculus (whith a polynomial bound, up to sharing). If we could generalize our technique to System F, we would obtain an alternative, and *constructive* proof of Strong Normalization (and maybe, with a quite low complexity bound).
The second major implication of our result is to show the actual expressiveness of sharing in higher order languages. Our polynomial bound should be compared with Schwichtemberg "tower of exponential" [Sc82]. Moreover, the encoding of SAT in the simply typed λ-calculus shows a quite interesting correspondence between *sharing* and *non-determinism* that was far from evident, and surely deserves further investigation. In particular, our result seems to open a completely innovative perspective on the characterization of complexitity classes in terms of typed λ-calculi.
From the point of view of optimality, we have many implications on the actual complexity of higher order sharing. In [As96] we proved that Lamping's algorithm could be exponential w.r.t. the number of families of the term. Now, we also understand why we could not reasonably expect a better result. Indeed, an obvious corollary of the fact that $\boldsymbol{P = NP}$ up to sharing is:

Proposition 5.1 *If $\boldsymbol{P \neq NP}$ then, in general, the computational overhead of (optimal) sharing in the reduction of a term $\boldsymbol{M}$ cannot be polynomial in the number of its families*[11]*.*

On the other side, (if $\boldsymbol{P \neq NP}$) the number of family reduction is definitely not a good measure of the "intrinsic" complexity of λ-terms (although the hypothesis $\boldsymbol{P \neq NP}$ suggests that it could be quite difficult to give a direct and formal proof of this claim). Criticisms against this measure had been already raised by many authors [As96, ML96, ML97]. In particular, the total number of interactions in Lamping's "abstract" system looks as a much more realistic bound to the "intrinsic complexity" of λ terms (note that, essentially, the expansion technique consists in transforming β-interactions into fan-annihilations).
Let us finally remark that our result does not diminish the interest of Lamping's technique. On the contrary! In [As96] we conjectured that, under a few assumptions not worth recalling here, the total number of interactions for reducing a term $\boldsymbol{M}$ in Lamping's algorithm is at most 2^f, if $\boldsymbol{f}$ is the number of families in $\boldsymbol{M}$. This has still to be compared against Schwichtenberg elementary bound (a tower of exponential).

References

[As96] A. Asperti. *On the complexity of β-reduction.* Proceedings of the twenty-third Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'96).

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

[10] Up to sharing, of course.

[11] We have been tempted to simply write this statement in the form "the problem of optimal sharing is NP-hard". However, it is not so clear what "the problem of sharing" actually means.

[Laf90] Y. Lafont. *Interaction Nets.* Proc. of the 17th Symposium on Principles of Programming Languages (POPL 90). San Francisco. 1990.

[Lam90] J. Lamping. *An algorithm for optimal lambda calculus reductions.* Proc. of the 17th Symposium on Principles of Programming Languages (POPL 90). San Francisco. 1990.

[Le78] J.J.Levy. *Réductions correctes et optimales dans le lambda-calcul.* Thèse de doctorat d'état, Université de Paris VII. 1978.

[ML96] H. Mairson, J. L. Lawall. *Optimality and Inefficiency: What Isn't a Cost Model of the Lambda Calculus?* ACM International Conference on Functional Programming, pp. 92–101. 1996.

[ML97] H. Mairson, J. L. Lawall. *On the global dynamics of optimal graph reduction.* 1997 ACM International Conference on Functional Programming. 1997.

[Sc82] H. Schwichtenberg. *Complexity of normalization in the pure typed lambda calculus.* In: *The L.E.J.Brower Centenary Symposium*, A.S.Troelstra, D. Van Dalen eds., North-Holland, Amsterdam. 1982.