

# What is an Efficient Implementation of the $\lambda$ -calculus?

*Gudmund S. Frandsen*<sup>1 2 3</sup>  
*Department of Computer Science*  
*Aarhus University*

*Carl Sturtivant*  
*Department of Computer Science*  
*University of Minnesota*

version January 31, 1991

---

<sup>1</sup>This research was partially carried out, while visiting Dartmouth College, New Hampshire.

<sup>2</sup>This research was partially supported by the Danish Natural Science Research Council (grant No. 11-7991).

<sup>3</sup>This research was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (project ALCOM).

## Abstract

We propose to measure the efficiency of any implementation of the  $\lambda$ -calculus as a function of a new parameter  $\nu$ , that is itself a function of any  $\lambda$ -expression.

Complexity is expressed here as a function of  $\nu$  just as runtime is expressed as a function of the input size  $n$  in ordinary analysis of algorithms. This enables implementations to be compared for worst case efficiency.

We argue that any implementation must have complexity  $\Omega(\nu)$ , i.e. a linear lower bound. Furthermore, we show that implementations based upon Turner Combinators or Hughes Super-combinators have complexities  $2^{\Omega(\nu)}$ , i.e. an exponential lower bound.

It is open whether any implementation of polynomial complexity,  $\nu^{O(1)}$ , exists, although some implementations have been implicitly claimed to have this complexity.

# Introduction

## Objectives

The aim of this paper is to provide a theoretical basis for efficiency considerations in the implementation of functional languages.

So far, people working in this area have approached the subject as programmers. That is to say, they have taken an intuitive approach to efficiency backed up by empirical trials once some new implementation has been created (e.g. measuring the runtimes of benchmark programs). Their major concerns have been to increase the expressive power of functional languages and to create more efficient implementations so that functional languages are usable.

From our standpoint, the implementation problem for functional languages is just another algorithmic problem, albeit one of some difficulty. In other areas of algorithm design, there is a sound theoretical framework: the notion of input size,  $n$ ; measuring the worst case and average case runtimes of an algorithm on inputs of size  $n$  in  $O$ -notation; and the idea of an optimal algorithm as one whose runtime as a function of  $n$  is of minimum growth. Such tools are not presently available to implementers of functional languages. Whilst it is not necessarily true that theoretically “fast” algorithms are the best in practice, nevertheless an investigation of various theoretically fast algorithms provides a good starting point for practical implementation considerations.

It is our aim to provide an analogous quantitative theoretical framework in which to assess the complexities of implementations of functional languages. This would ensure that the same tools are available to implementers of functional languages as are available to other algorithm designers.

Once such a framework is in place, we will be able to define the notion of an optimal implementation. This has importance in understanding what the complexity of a functional program is, as is discussed next.

## The Complexity of a Functional Program

Currently, it is not known how to analyse the complexity of a functional program in an implementation independent way. Indeed, it is not clear that the complexity of a functional program is in any sense implementation indepen-

dent. This is in marked contrast to programs written in common imperative languages. Here it is informally understood that all good implementations endow any given program with essentially the same complexity (in  $O$ -notation).

Clearly, there are bad implementations of imperative languages in which the complexities of some programs are degraded below their “true” complexities. Thus the reason that the complexity of programs in imperative languages is well defined is that we have identified the good implementations (i.e. optimal implementations in  $O$ -terms), and we regard these implementations as defining *the* complexity of a program. Consequently, the complexity of an imperative program is implicitly implementation independent in this sense.

If we wish to arrive at a similar situation in regard to the complexity of functional programs, then we need to know what an optimal implementation of a functional language is.

We introduce a complexity theoretic definition of the efficiency of any implementation of a functional language. Using this definition, we define what an optimal implementation is. A basic acquaintance with the lambda calculus is assumed[Chu32, Ros84, Bar81].

The first half of the paper is concerned with philosophical and motivational considerations, and gives an informal discussion of the technical results given in full detail in the second half of the paper.

## Philosophical Considerations

In this paper we propose to consider the problem of implementing the pure  $\lambda$ -calculus, which may be regarded as a (rather difficult to use) functional programming language. We give a complexity theoretic framework for considering the efficiency of such implementations. We argue below why this then provides a complexity theoretic framework for understanding the efficiency of implementations of *any* functional language. Hereafter, we will use the phrase “*lambda-calculus*” to mean the pure *lambda-calculus*.

## Why Study the Pure Lambda Calculus?

The reader may ask how it is that pure lambda expressions can reflect the computational power of their favorite functional languages. In particular,

the reader may say that the reader's favorite functional language has a lot of added primitives (functions, data structures, etcetera) that are not present in the pure lambda calculus, and thus why are results about the pure lambda calculus applicable to these extensions?

Our reply is an argument to establish informally that these features may be encoded into the pure  $\lambda$ -calculus in such a way that there is an implementation of the pure  $\lambda$ -calculus endowing these features with a certain complexity if and only if there is an implementation of the functional language directly endowing the features with the same complexity (up to a constant factor). The argument proceeds as follows:-

We fix some machine model of sequential computation (there are some technical difficulties with the RAM model of computation: see the appendix for a complete discussion of this argument). Then we give an encoding of any machine in this model as a pure  $\lambda$ -expression, with the property that a constant number of  $\beta$ -reduction steps simulate an atomic step of the machine and these involve only a bounded amount of work. (The constant may depend upon the machine being encoded. In fact, it can be arranged that the expression encoding a machine follows exactly the steps that the machine would take if a normal order reduction strategy is used.) Thus particular functions and data structures may be embedded naturally in the pure  $\lambda$ -calculus along with their imperative implementations.

At this point, the reader may agree that many of the features of the reader's favorite functional language can be encoded in the pure  $\lambda$ -calculus in such a way as to retain the potential for efficient implementation. Consider an arbitrary program in the reader's favorite functional language. Even if we strip away all its syntactic sugar of the above forms, and replace it with  $\lambda$ -encodings, what remains may not be a simple  $\lambda$ -expression. The reader may say that what remains is what may be termed "a mutually recursive  $\lambda$ -program": i.e. a series of mutually recursive definitions in the pure  $\lambda$ -calculus. (In fact there may be some additional control structures, etcetera, but hopefully the reader will be convinced that these can be encoded as simple  $\lambda$ -expressions whilst retaining the possibility of equally efficient implementation.)

A careful investigation of the sizes of the expressions in the multiple fixed point theorem ([Bar81] p.142) shows that such a mutually recursive  $\lambda$ -program defines a  $\lambda$ -expression whose size is linear in the size of the original program. Thus an efficient implementation of  $\lambda$ -expressions would provide

an efficient implementation of  $\lambda$ -programs. (We have not discussed the input to the functional program — we assume that it is a part of the program here. For further discussion of this point see below.)

We have given an informal argument why any functional programming language may be encoded as pure  $\lambda$ -expressions whilst maintaining the possibility of equally efficient implementation. Without a formal definition of a functional language, it is not possible to prove our contention. For this reason we refer to it as the *implementation thesis* for functional languages.

### **The Implementation Thesis:**

Any functional programming language may be efficiently compiled into the pure  $\lambda$ -calculus in such a way as to retain the possibility of equally efficient implementation.

The converse of this thesis is also a matter of definition: any functional language has the power to directly define an arbitrary pure  $\lambda$ -expression—one may take this as a part of the definition of a functional language. (N.B. Presumably some strongly typed languages do not have this property—these are possibly weaker than the pure  $\lambda$ -calculus in computational power, and this may improve their implementability.) In any case we have argued that the efficient implementation problem for functional languages is at most as difficult as the implementation problem for pure  $\lambda$ -expressions (in a theoretical sense), and for those functional languages that contain the pure  $\lambda$ -calculus as a sub-language, the problems are equally hard.

One detail that has been glossed over is that real programs tend to be separated from their inputs. By using pure  $\lambda$ -expressions as the canonical model of a functional language we lose this. The program and input (if any) are both  $\lambda$ -expressions, and we consider the cost of reducing the expression consisting of the first applied to the second (or just the first). However, this just amounts to taking the cost of “compilation” and the cost of “run” together, and is no disadvantage from a theoretical standpoint. (Of course in practice a program may be compiled and used on many different inputs. We regard this as a different issue, and ignore it.)

In the case of an interactive program, not all of the input is available as the run proceeds. This restricts the implementation possibilities, and therefore can only make life harder than the problem we consider.

## The Pure $\lambda$ -Calculus

In this paper we propose to consider the following problem, that we call  $\lambda$ -expression normal form computation (LENF). Input to LENF is a  $\lambda$ -expression in the standard syntax, and the corresponding output is the normal form of that expression if such exists, and otherwise is empty (a solution being allowed not to halt in this case).

Currently there seems to be a belief that there is neither a canonical way of simulating reduction in the  $\lambda$ -calculus efficiently nor a canonical way to define formally what efficiently should mean. This is best illustrated by a citation:

*There have been some studies about the relative efficiencies of various implementation techniques for applicative languages, but there are no clear winners. This should not be surprising at all, if we consider the generality of the problem. We are dealing with the efficiency of the process of evaluating arbitrary partial recursive functions. Standard complexity theory is clearly not applicable to such a broad class of computable functions. The time-complexity of such a universal procedure cannot be bounded by any computable function of the size of the input. ... Complexity theory is concerned with the inherent difficulty of the problems (or classes of problems) rather than the overall performance of some particular model of a universal computing device. [R  v88, page 131]*

Furthermore, the result that no strategies that choose optimal reduction sequences are recursive [Bar81, ch. 13] may mislead if not examined closely, because of its dependence upon  $\lambda$ -expressions with no normal form. In fact the problem

*Input:* A  $\lambda$ -expression  $M$  with a normal form

*Output:* A minimal length reduction sequence to normal form for  $M$

has an algorithmic solution: simply explore all reduction sequences that use up to the number of reductions which the leftmost strategy takes to reach normal form, and choose a minimal one.

We do not claim that this procedure is efficient; however, its runtime is bounded by some recursive function of the number of  $\beta$ -reductions in the normal order reduction sequence to normal form. Furthermore, this is also

true if we extend the input set to include all  $\lambda$ -expressions, since those with no normal form have infinite normal order reduction sequences.

This suggests that a change of the definition of “input size” to include some measure of the length of reduction chains may make it possible to bound the runtime of an implementation of LENF by some recursive function of the new input size. This would also hold in the case of  $\lambda$ -expressions with no normal form, since their input size would be infinity. This is the approach we take. In the next section we discuss our precise definition of the new input size.

## Definition of the New Input Size Parameter

Lévy proposed an optimal implementation to be one in which “work is not duplicated”. He has given a formal definition in terms of a parallel reduction step for a labelled version of the  $\lambda$ -calculus [Lév80, Lév88]. We do not take up his notion of optimality since our aim is not to minimise the number of some kind of reduction steps, but rather to minimise the overall runtime. However, we do adopt his notion of parallel reduction.

Let  $L$  be any  $\lambda$ -expression. We define  $\mu_\pi(L)$  to be the minimum number of parallel reduction steps required to reduce  $L$  to normal form (infinity if  $L$  has no normal form).

We define the new input size parameter  $\nu$  of the  $\lambda$ -expression  $L$  with normal form  $N$  to be the size of  $L$  plus the size of  $N$  plus  $\mu_\pi(L)$  ( $\nu = \infty$  if  $L$  has no normal form).

The philosophy behind this definition is as follows—We wish to put a complexity measure on  $\lambda$ -expressions and to do this we must arrange the definition of the input size parameter  $\nu$  so that it conforms with the *bounded runtime principle*; namely that there exists an implementation whose runtime is bounded above by some recursive function of  $\nu$ . (Recursive functions are considered to be extended to map infinity to infinity.)

The size of  $L$  must appear in  $\nu$  because if the input is already in normal form, then this must be verified by any implementation. In general, this must take time proportional to the size of  $L$ .

A measurement of the length of some sort of canonical reduction chain to normal form must be included in  $\nu$  because otherwise expressions with no normal form constitute a family of inputs in conflict with the bounded run-

time principle. Lambda-expressions defining arbitrarily complex functions will also give violations of the bounded runtime principle if the input size parameter does not involve such a measure of reduction. We choose  $\mu_\pi$  as this measure. Some other candidates that spring to mind are the minimum number of  $\beta$ -reductions to reach normal form,  $\mu_\beta$ , or the length of the normal order reduction sequence to normal form,  $l_{normal}$ . (We ignore  $\eta$  reduction here; our justification stems from corollary 15.1.6 and the first part of the proof of corollary 15.1.5 in [Bar81].)

Finally, the length of the normal form  $N$  should be present since there are many families of  $\lambda$ -expressions with the property that the size of the normal form is exponential in the number of reduction steps to normal form. (This is analogous to an algorithm whose output size is exponential—the runtime is exponential for trivial reasons.) Thus the absence of the size of  $N$  would automatically imply an exponential lower bound on all implementations: we would like a definition that does not impose such trivial limitations.

These three arguments justify most of the definition of  $\nu$ , excepting the choice of  $\mu_\pi$  as a measurement of reduction chain length.

Of the two other obvious candidates given above ( $\mu_\beta$  and  $l_{normal}$ ),  $l_{normal}$  is not remotely optimal for some  $\lambda$ -expressions. Later in the paper (proposition 3) we give an example of a family of  $\lambda$ -expressions for which applicative order gives the minimum length reduction chain to normal form. For this family, Normal order reduction gives exponentially longer chains to normal form. Thus, measuring the complexity of an implementation using  $l_{normal}$  can give unrealistically optimistic assessments. For this reason, we rule out  $l_{normal}$ .

We choose  $\mu_\pi$  rather than  $\mu_\beta$  for a number of reasons. First, it is known that  $\mu_\pi$  is always less than or equal to  $\mu_\beta$ . Second, an optimal strategy is known for parallel reductions—normal order [Lév80], whereas none such is known for the optimal  $\beta$ -reduction sequence. Thus, by using  $\mu_\pi$  we get a harsher measure of complexity; however, this is nevertheless consistent with the bounded runtime principle: a naive simulation of the normal order parallel reduction sequence to normal form has a runtime that is bounded by a recursive function of  $\nu$ , as the reader may verify.

Furthermore  $\mu_\pi$  and  $\mu_\beta$  may be significantly different. Until recently we knew of no  $\lambda$ -expression for which they differ by more than a factor of three. However, in the second part of the paper, we give an infinite family of  $\lambda$ -expressions which seems to have the property that  $\mu_\beta$  is significantly greater than  $\mu_\pi$ . In particular, the third member of the family has  $\mu_\beta$  greater than

five times  $\mu_\pi$ . We conjecture for this family that  $\mu_\beta$  is not bounded by any constant multiple of  $\mu_\pi$ . Whether  $\mu_\beta$  can be as much as exponentially different from  $\mu_\pi$ , we do not know, but we think this may well be so.

The interesting positive factor about using parallel reductions instead of  $\beta$ -reductions is that parallel reduction was defined as a kind of improved  $\beta$ -reduction in which work was not duplicated (in the sense of doing several reductions instead of one because of copying of redices). To hope for an implementation of the  $\lambda$ -calculus of low (say polynomial  $\nu^{O(1)}$ ) complexity, is therefore to take an optimistic view that perhaps the duplication of work that sometimes seems unavoidable in ordinary  $\beta$ -reduction is unnecessary in a general implementation.

Thus we optimistically adopt  $\mu_\pi$ , whilst not being dogmatic in our choice. Should it transpire that all implementations must have very high complexity under this definition of input size  $\nu$ , then it may be useful to change to a larger notion of input size  $\nu$ . However, such a result would in itself be a great step forward in our understanding of the inherent complexity of implementations of the  $\lambda$ -calculus, and would seem to justify an initial choice of  $\mu_\pi$ .

## The Complexity of an Implementation

Now that we have defined the input size parameter  $\nu$ , it remains to define the complexity of an implementation  $\mathcal{I}$ , in the obvious way. Informally, the complexity of  $\mathcal{I}$  is just the time taken by  $\mathcal{I}$  to compute the normal form of a  $\lambda$ -expression, expressed as a function of  $\nu$ . However, as in everyday analysis of algorithms, this conceals the important choice of whether we take worst-case or some sort of average-case complexity to be the fundamental measure.

Just as in the usual setting, a problem with average-case complexity is the question of which probability distribution one should use over inputs of a given size. This problem is particularly pronounced in the case of implementations of the  $\lambda$ -calculus, because it may be important to focus on large sub-classes of expressions. For example, the kind of arguments given in the section on philosophical considerations, as to how various features of some functional programming language may be efficiently encoded in the pure  $\lambda$ -calculus, would certainly lead to a bizarrely biased collection of  $\lambda$ -expressions, depending very much upon the features of that language. Merely choosing

$\lambda$ -expressions of the form “program” applied to “input” would also bias the distribution. Thus naive assumptions such as taking the uniform distribution over  $\lambda$ -expressions with a given  $\nu$ , are likely to be meaningless.

For that reason, we take the worst-case complexity as the canonical measure of the complexity of an implementation of the  $\lambda$ -calculus.

Having made the choice to use the worst-case complexity, we then secure the benefit that if the implementation  $\mathcal{I}$  has been proven to have some complexity as a function of  $\nu$  then there are no “bad” families of inputs that violate this bound—we are absolutely assured that whatever we use the  $\lambda$ -calculus for, in whatever encoding, we will obtain the guaranteed performance.

In particular, suppose we have a  $\lambda$ -expression consisting of a “program” applied to an “input” (perhaps encoded as a binary list), where the input is of size  $n$ . If we now obtain the normal form of this expression using the implementation  $\mathcal{I}$ , then the worst case complexity of  $\mathcal{I}$  provides for a relationship between the conventional complexity expressed as a function of  $n$  and the complexity of the implementation.

# Summary of Technical Results and Open Problems

In the second part of the paper, we define formally the input size parameter  $\nu$  and the complexity of an implementation of the  $\lambda$ -calculus, in accordance with the principles discussed above. We argue that any implementation must have complexity  $\Omega(\nu)$ . Furthermore, we give lower bounds of  $2^{\Omega(\nu)}$  on implementations based upon Turner combinators or Hughes super-combinators.

We give a  $\lambda$ -expression for which  $\mu_\beta$  is greater than five times  $\mu_\pi$ . This  $\lambda$ -expression is the third member of an infinite family of  $\lambda$ -expressions for which  $\mu_\beta$  seems significantly greater than  $\mu_\pi$ . However, we do not know the values of  $\mu_\beta$  or  $\mu_\pi$  for all of this family. The problem of finding a bound for  $\mu_\beta$  in terms of  $\mu_\pi$  in general (i.e. that holds for all  $\lambda$ -expressions) remains unresolved.

Many other implementations of the  $\lambda$ -calculus exist that have not been analysed. Furthermore, some authors ([Sta82, Lam90]) have claimed their implementations optimal in the sense of Lévy, (i.e. that they do not simulate more than  $\mu_\pi$  reduction steps). However, optimality in this sense does not impose significant constraints on the complexity of an implementation. Hence, the existence of a polynomial time ( $\nu^{O(1)}$ ) implementation remains an open question.

## Preliminaries and Definitions

### Definition 1

#### $\lambda$ -expressions

The set of  $\lambda$ -expressions,  $\Lambda$ , is defined inductively,

- variables:  $x_i \in \Lambda$  for all  $i \in \{0, 1, 2, 3, \dots\}$
- abstraction: if  $l \in \Lambda$  then  $(\lambda.l) \in \Lambda$
- application: if  $l_1, l_2 \in \Lambda$  then  $(l_1 l_2) \in \Lambda$

We adopt the convention that

$$l_1 l_2 l_3 \dots l_n = (\dots ((l_1 l_2) l_3) \dots l_n)$$

and let  $I$  denote the identity expression:

$$I = (\lambda.x_0)$$

In the original syntax for  $\lambda$ -expressions [Chu32], the variable bound in the abstraction is written explicitly in the prefix, e.g.  $\lambda x.l$  denotes that the specific variable  $x$  is bound. However, we adopt deBruijn's convention for naming variables [deB72] according to which an occurrence of  $x_i$  denotes the variable that is bound at the  $(i + 1)$ 'th enclosing abstraction (if such exists). Hence it is unnecessary to specify a variable name in the abstraction prefix.

As an example consider the least fixed point operator,

$$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

as described with named variables. In deBruijn's notation it is

$$\lambda.(\lambda.x_1(x_0x_0))(\lambda.x_1(x_0x_0)).$$

Both notations are described in [Bar81]. The deBruijn-syntax is often preferable for formal reasoning since name clashes do not arise (and  $\alpha$ -reduction becomes obsolete), whereas the syntax with names facilitates human perception. We use either of the two notations at our discretion.

## Definition 2

### $\beta$ -reduction

- A *redex* is a  $\lambda$ -expression of the form  $(\lambda.l_1)l_2$ .
- A specific occurrence of a variable  $x_i$  in the expression  $l$  is *free* if this occurrence has less than  $i + 1$  enclosing abstractions.
- A redex may be transformed in a  $\beta$ -reduction

$$(\lambda.l_1)l_2 \rightarrow_{\beta} l'_1$$

where  $l'_1$  is formed from  $l_1$  by the following sequence of transformations.

1. All occurrences of the abstracted variable (i.e. those  $x_i$  in  $l_1$  that have precisely  $i$  surrounding abstractions in  $l_1$ ), are replaced by specially modified versions of  $l_2$ , viz. in the replacement of  $x_i$ , free variables of  $l_2$  have their indices incremented by  $i$  to reflect the modified number of enclosing abstractions.

2. All other free occurrences of variables in  $l_1$  have their index decremented by 1.
- Let  $l = l_1[r]$  denote that the  $\lambda$ -expression  $l$  contains the subexpression  $r$  in the context  $l_1$ . If  $l = l_1[r]$  and  $r \rightarrow_\beta r'$  then  $l_1[r] \rightarrow_\beta l_1[r']$ .
  - The notation  $\rightarrow_\beta^*$  denotes the reflexive and transitive closure of  $\rightarrow_\beta$ .
  - A  $\lambda$ -expression that contains no redex is in *normal form*. A *normal order* reduction sequence  $l_0 \rightarrow_\beta l_1 \rightarrow_\beta \dots \rightarrow_\beta l_n$  is one in which the leftmost outermost redex is chosen for transformation at each step.
  - A reduction sequence  $l_0 \rightarrow_\beta l_1 \rightarrow_\beta \dots \rightarrow_\beta l_n$ , where  $l_n$  is in normal form, is *complete* and  $l_n$  is the normal form of  $l_0$ .
  - The notation  $\mu_\beta(l)$  denotes the length of the shortest possible complete reduction sequence starting from  $l$ . If no such sequence exists the value is  $\infty$ .

By the Church Rosser property (simple proof in [Ros84]), the normal form of a  $\lambda$ -expression is unique. If an expression  $l$  has a normal form then a normal order reduction sequence will eventually lead to it, but not necessarily in the smallest possible number of steps.

## Definition 3

### Parallel Reductions

- The set of *labels*  $L$  is defined in terms of an infinite set of atomic labels  $A$ .
  1.  $A \subseteq L$
  2. If  $u, v \in L$  then  $uv \in L$
  3. If  $u \in L$  then  $\underline{u} \in L$
- A *labelled*  $\lambda$ -expression has a label attached to every subexpression. In an *initial* expression all labels are atomic and mutually different.
- In a  $\beta$ -reduction labels are modified according to the following rule ( $M, N$  denote  $\lambda$ -expressions and  $u, v$  denote labels):

$$1. ((\lambda x.M)^u N)^v \rightarrow_\beta (M^u[x \rightarrow N^u])^v$$

Multiple labels are concatenated into a single label:

$$2. (M^u)^v \rightarrow M^{uv}$$

(When using the named  $\lambda$ -calculus, we assume that renaming has occurred so as to avoid name clashes).

- Given two labelled  $\lambda$ -expressions  $l_0, l_1$  such that  $l_0$  is initial and  $l_0 \rightarrow_\beta^* l_1$ , we say that two redices  $r_1, r_2$  in  $l_1$  are *equivalent* with respect to  $l_0$  if they have identical labels, i.e. if  $r_1 = ((\lambda x.M_1)^{u_1} N_1)^{v_1}$  and  $r_2 = ((\lambda y.M_2)^{u_2} N_2)^{v_2}$  then  $u_1 = u_2$
- A *parallel* reduction sequence  $l_0 \rightarrow_\pi l_1 \rightarrow_\pi \dots \rightarrow_\pi l_n$  is a sequence in which all redices in precisely one equivalence class with respect to  $l_0$  are reduced in a single step. (The ordinary reductions in this step are done from the bottom up until no more reductions with the same label are possible. This process always terminates.)
- A parallel reduction sequence is *normal order* if in each step the equivalence class being reduced contains the syntactically leftmost redex.
- A *complete* parallel reduction sequence ends with a  $\lambda$ -expression in normal form.
- The notation  $\mu_\pi(l)$  denotes the length of the shortest complete parallel reduction sequence starting from  $l$ . If no such sequence exists the value is  $\infty$ .

Two equivalent redices may overlap syntactically, so the notion of a parallel reduction step must be defined with great care. See [Lév80] for details. Our condensed definition is due to [Klo80] and described in [Fie90]. The original investigation of parallel reduction is due to Lévy [Lév80, Lév88].

Intuitively a parallel reduction sequence avoids duplication of work. All “identical” redices are reduced in a single step, where two redices are “identical” if for some regular reduction sequence they arise as copies (i.e. *residuals*) of a single redex.

It is known that a complete parallel normal order reduction sequence for a  $\lambda$ -expression  $l$  has length  $\mu_\pi(l)$  (i.e. normal order is an optimal strategy) and  $\mu_\pi(l) \leq \mu_\beta(l)$  for all  $\lambda$ -expressions  $l$  [Lév80].

## Definition 4

### The Complexity of an Implementation of the $\lambda$ -calculus.

- A procedure  $\mathcal{I}$  is an *implementation* of the  $\lambda$ -calculus if  $\mathcal{I}$  on input a  $\lambda$ -expression  $M$  outputs  $N$ , the normal form of  $M$  (if  $M$  has no normal form then  $\mathcal{I}$  need not halt).
- The *input-size* parameter  $\nu : \Lambda \rightarrow \mathbf{N} \cup \{\infty\}$  is given by

$$\nu(M) = |M| + \mu_\pi(M) + |N|$$

for all  $M \in \Lambda$  where  $N$  is the normal form of  $M$ , and the norm denotes expression size. (If  $M$  has no normal form then  $\nu(M) = \infty$ ).

- An implementation  $\mathcal{I}$  of the  $\lambda$ -calculus has *worst-case complexity*  $T(\nu)$  if  $T : \mathbf{N} \rightarrow \mathbf{N}$  satisfies

$$T(\nu) = \text{Max}\{\text{run time of } \mathcal{I} \text{ on input } M \mid M \in S_\nu\}$$

when  $S_\nu \neq \emptyset$  and where  $S_\nu = \{M \in \Lambda \mid \nu(M) = \nu\}$ .

- An implementation  $\mathcal{J}$  of the  $\lambda$ -calculus of worst case complexity  $T_{\mathcal{J}}(\nu)$ , is an *optimal* implementation if for any implementation  $\mathcal{I}$  of the  $\lambda$ -calculus of worst case complexity  $T_{\mathcal{I}}(\nu)$ , we have  $T_{\mathcal{J}}(\nu) = O(T_{\mathcal{I}}(\nu))$ .

The complexity of an implementation is well defined, since the sets  $S_\nu$  are always finite, and there is a number  $\nu_0$  such that for  $\nu > \nu_0$ ,  $S_\nu$  is always non empty.

## A Linear Lower Bound

### Proposition 1

**Any implementation of the  $\lambda$ -calculus has complexity  $\Omega(\nu)$ .**

*Outline Proof:* In the appendix we give an encoding of an arbitrary (imperative) computation as a  $\lambda$ -expression with the property that  $\mu_\pi = \mu_\beta = l_{\text{normal}}$ , and each step of the computation is simulated by a constant number of reductions which in turn can be implemented in such a way as to take only

a bounded amount of work. Thus the existence of an implementation with complexity  $o(\nu)$  would immediately give rise to a method of speeding up an arbitrary computation. QED.

## Exponential Lower Bounds for some Combinator-based Implementations

### Proposition 2

**Implementations of the  $\lambda$ -calculus based on Turner combinators or Hughes super combinators are exponentially inefficient.**

*Proof:* We shall exhibit a family of  $\lambda$ -expressions for which both a translation to Turner combinators and a translation to Hughes super combinators results in exponentially inefficient executions.

The construction will exploit the fact that a combinator may take more than one argument, and the reduction rules for such a combinator can not be used unless all arguments are present. In contrast the original  $\lambda$ -expression can be partially evaluated, since each abstraction  $(\lambda x.M)$  refers to only one variable.

Define a family  $\{A_n\}$  of  $\lambda$ -expressions by

$$A_0 = (\lambda x.I)$$

$$A_n = (\lambda h.(\lambda w.wh(ww))A_{n-1}) \text{ for } n \geq 1$$

By induction we may prove that  $A_n \rightarrow_\beta^{4^n} A_0$ . The case  $n = 0$  is obvious. The induction step is proved by considering the following sequence of reductions:

$$\begin{aligned} A_n &= (\lambda h.(\lambda w.wh(ww))A_{n-1}) \\ &\rightarrow_\beta^{4(n-1)} (\lambda h.(\lambda w.wh(ww))A_0) \\ &\rightarrow_\beta (\lambda h.A_0h(A_0A_0)) \\ &\rightarrow_\beta (\lambda h.I(A_0A_0)) \\ &\rightarrow_\beta (\lambda h.A_0A_0) \\ &\rightarrow_\beta (\lambda h.I) \\ &= A_0 \end{aligned}$$

The exponentially bad behaviour will be obtained for the family  $\{B_n\}$ , where  $B_n = A_n I$ . We see that  $\mu_\beta(B_n) \leq 4n + 1$  using the result above. Hence  $\nu \leq |B_n| + \mu_\beta(B_n) + |I| = O(n)$

Let us first consider a translation to Turner combinators. The following 8 combinators are used:

$$\begin{aligned}
\mathbf{S}abc &= ac(bc) \\
\mathbf{K}ab &= a \\
\mathbf{I}a &= a \\
\mathbf{B}abc &= a(bc) \\
\mathbf{C}abc &= acb \\
\mathbf{S}'abcd &= a(bd)(cd) \\
\mathbf{B}'abcd &= ab(cd) \\
\mathbf{C}'abcd &= a(bd)c
\end{aligned}$$

The combinators  $\mathbf{S}$  and  $\mathbf{K}$  alone suffice, but to avoid any critique saying that the exponentially bad behaviour was caused by not using a specific combinator, we include all 8 above.

The translation process is described by rewrite rules:

1.  $(\lambda x.x) \rightarrow \mathbf{I}$   
 $(\lambda x.y) \rightarrow \mathbf{K}y$  for  $y$  being a variable  $y \neq x$ .  
 $(\lambda x.MN) \rightarrow \mathbf{S}(\lambda x.M)(\lambda x.N)$
2. The following rules should be used whenever they apply to some intermediate expression generated by the rules in (1) ( $A, B$  are arbitrary expressions):

$$\begin{aligned}
\mathbf{S}(\mathbf{K}A)(\mathbf{K}B) &\rightarrow \mathbf{K}(AB) \\
\mathbf{S}(\mathbf{K}A)\mathbf{I} &\rightarrow A \\
\mathbf{S}(\mathbf{K}A)B &\rightarrow \mathbf{B}AB \\
\mathbf{S}A(\mathbf{K}B) &\rightarrow \mathbf{C}AB
\end{aligned}$$

3. Whenever an intermediate expression is formed to which one of the following rules apply, use it: ( $A, B, K$  are arbitrary expressions)

$$\begin{aligned}
\mathbf{S}(\mathbf{B}KA)B &\rightarrow \mathbf{S}'KAB \\
\mathbf{B}(KA)B &\rightarrow \mathbf{B}'KAB \\
\mathbf{C}(\mathbf{B}KA)B &\rightarrow \mathbf{C}'KAB
\end{aligned}$$

The rules (1) suffice, but the additional rules makes the generated code more efficient for some  $\lambda$ -expressions.

The  $\lambda$ -expression  $\{B_n\}$  translates into combinator expressions  $\{\mathcal{B}_n\}$  that may be described recursively:

$$\begin{aligned}\mathcal{A}_0 &= \mathbf{KI} \\ \mathcal{A}_n &= \mathbf{C}(\mathbf{C}'\mathbf{S}(\mathbf{CI})(\mathbf{SII}))\mathcal{A}_{n-1} \text{ for } n \geq 1 \\ \mathcal{B}_n &= \mathcal{A}_n\mathbf{I}\end{aligned}$$

Let  $S(n)$  denote the minimal number of combinator reduction steps that brings  $\mathcal{A}_n x$  to its normal form  $\mathbf{I}$  (independent of expression  $x$ ). Clearly  $S(0) = 1$  and  $S(n)$  may be found by an inductive argument:

$$\begin{aligned}\mathcal{A}_n x &= \mathbf{C}(\mathbf{C}'\mathbf{S}(\mathbf{CI})(\mathbf{SII}))\mathcal{A}_{n-1} x \\ &\rightarrow \mathbf{C}'\mathbf{S}(\mathbf{CI})(\mathbf{SII})x\mathcal{A}_{n-1} \\ &\rightarrow \mathbf{S}(\mathbf{CI}x)(\mathbf{SII})\mathcal{A}_{n-1} \\ &\rightarrow \mathbf{CI}x\mathcal{A}_{n-1}(\mathbf{SII}\mathcal{A}_{n-1}) \\ &\rightarrow \mathbf{I}\mathcal{A}_{n-1}x(\mathbf{SII}\mathcal{A}_{n-1}) \\ &\rightarrow \mathcal{A}_{n-1}x(\mathbf{SII}\mathcal{A}_{n-1}) \\ &\rightarrow^{S(n-1)} \mathbf{I}(\mathbf{SII}\mathcal{A}_{n-1}) \\ &\rightarrow \mathbf{SII}\mathcal{A}_{n-1} \\ &\rightarrow \mathbf{I}\mathcal{A}_{n-1}(\mathbf{I}\mathcal{A}_{n-1}) \\ &\rightarrow \mathcal{A}_{n-1}(\mathbf{I}\mathcal{A}_{n-1}) \\ &\rightarrow^{S(n-1)} \mathbf{I}\end{aligned}$$

The above reduction is optimal because we are forced to copy the complex expression for  $\mathcal{A}_{n-1}$  since it is irreducible (though extensionally equivalent to  $\mathcal{A}_0$ ). From the recurrence relation  $S(n) = 2S(n-1) + 8$  and the initial condition above, we obtain  $S(n) = 9 \cdot 2^n - 8$ , which also is the number of steps necessary to reduce  $\mathcal{B}_n$ . Hence we conclude that an implementation of the  $\lambda$ -calculus based on Turner combinators has complexity  $2^{\Omega(n)}$ .

Next we consider super combinators. Instead of using a fixed set of combinators we customise “super” combinators to the specific  $\lambda$ -expression at hand. In the translation process we first eliminate occurrences of free variables within the body of a single  $\lambda$ -expression, by possibly binding more than one variable to a single  $\lambda$ . In  $B_n$  the only affected subexpression is  $(\lambda w.wh(w))$  that is converted into  $(\lambda w'w.wh'(w))h$ . For the translation of  $B_n$  the following  $n + 2$  super combinators will be formed  $(\mathbf{L}, \mathbf{M}_0, \mathbf{M}_1, \mathbf{M}_2, \dots, \mathbf{M}_n)$ :

$$\begin{aligned}
\mathbf{L}w'w &= ww'(ww) \\
\mathbf{M}_j h &= \mathbf{L}h\mathbf{M}_{j-1} \text{ for } 1 \leq j \leq n \\
\mathbf{M}_0 h &= \mathbf{I}
\end{aligned}$$

$B_n$  is translated into  $\mathbf{M}_n \mathbf{I}$ .

Let  $T(n)$  denote the number of reduction steps needed to reduce  $\mathbf{M}_n x$  to  $\mathbf{I}$  (independent of expression  $x$ ). Clearly  $T(0) = 1$ . By an inductive argument, we can find  $T(n)$ :

$$\begin{aligned}
\mathbf{M}_n x &\rightarrow \mathbf{L}x\mathbf{M}_{n-1} \\
&\rightarrow \mathbf{M}_{n-1}x(\mathbf{M}_{n-1}\mathbf{M}_{n-1}) \\
&\rightarrow^{T(n-1)} \mathbf{I}(\mathbf{M}_{n-1}\mathbf{M}_{n-1}) \\
&\rightarrow \mathbf{M}_{n-1}\mathbf{M}_{n-1} \\
&\rightarrow^{T(n-1)} \mathbf{I}
\end{aligned}$$

The order of reduction applied here is optimal, since we are forced to copy the irreducible combinator  $\mathbf{M}_{n-1}$ . From the recurrence relation  $T(n) = 2T(n-1) + 3$  and the initial condition above we obtain  $T(n) = 4 \cdot 2^n - 3$  and conclude that also super combinators lead to an implementation of complexity  $2^{\Omega(n)}$ . QED.

The combinators **S, K, I, B, C** were first defined by Schönfinkel [Sch24] under the names **S, C, I, Z, T**. Turner introduced the combinators **S', B', C'** [Tur79a]. Functional abstraction (the translation process) is described in [Sch24, CuFe58, Tur79a]. Application of combinators to functional programming is described in [Tur79b]. Super combinators and their application to functional programming are described in [Hug82, Hug84].

The exponential inefficiency relies in both cases on the existence of combinator expressions that are irreducible but extensionally equivalent to much simpler expressions. One may extend combinatory logic with axioms for extensionality [CuFe58, Bar81], but it seems unlikely that it would be computationally feasible to deal with extensionality in any combinator-based implementation of the  $\lambda$ -calculus such that this exponentially bad worst-case behaviour is eliminated.

Previous critiques of combinator-based translation have focused on a potentially quadratic size blow up in the translation from  $\lambda$ -expressions to Turner combinators and devised techniques to avoid this phenomenon [Bur82, Nos85, KeSl87]. Hughes proved that his super combinators have at

most a quasi-linear size blow up during translation from  $\lambda$ -expressions to combinators [Hug84].

## The Relation Between $l_{normal}$ and $\mu_\beta$

### Proposition 3

There is an infinite family  $\{A_k\}$  of  $\lambda$ -expressions such that  $l_{normal}(A_k) = 2^{\Omega(\mu_\beta(A_k))}$

.

*Proof:*

We construct the expressions  $A_k$  such that reduction in applicative order is exponentially more efficient than reduction in normal order.

Let

$$\begin{aligned} A_k &= (\lambda x.xx)A_{k-1} \\ A_0 &= I \end{aligned}$$

Let  $S(k)$  and  $T(k)$  denote the number of  $\beta$ -reduction steps needed to transform  $A_k$  to normal form using normal and applicative order respectively. It is easy to obtain the recurrence equations

$$\begin{aligned} S(k) &= 2 \cdot S(k-1) + 2 \\ S(0) &= 0 \\ T(k) &= T(k-1) + 2 \\ T(0) &= 0 \end{aligned}$$

that have solutions

$$\begin{aligned} S(k) &= 2^{k+1} - 2 \\ T(k) &= 2 \cdot k \end{aligned}$$

from the which the proposition follows.

## The Relation Between $\mu_\beta$ and $\mu_\pi$

### Proposition 4

There is a  $\lambda$ -expression  $A$  such that

$$\mu_\beta(A) > 5 \cdot \mu_\pi(A)$$

*Proof:* It will suffice to take  $A$  to be  $A_3$  in the following infinite family of  $\lambda$ -expressions:

$$\begin{aligned} A_0 &= I \\ A_1 &= (\Delta(\lambda h_1.h_1 I)) \\ A_2 &= (\Delta(\lambda h_2.\Delta(\lambda h_1.h_2(h_1 I)))) \end{aligned}$$

and in general

$$A_n = (\Delta(\lambda h_n.(\dots(\Delta(\lambda h_1.h_n(\dots(h_1 I)\dots)))\dots)))$$

where  $\Delta = (\lambda x.xx)$ .

Computer simulations show the following values

$$\begin{array}{rcccc} n : & 0 & 1 & 2 & 3 \\ \mu_\pi(A_n) : & 0 & 4 & 9 & 24 \\ \mu_\beta(A_n) : & 0 & 4 & 12 & 122 \end{array}$$

QED.

We have not been able to derive an explicit formula for  $\mu_\beta(A_n)$  or  $\mu_\pi(A_n)$ , but we conjecture that

$$\lim_{n \rightarrow \infty} \frac{\mu_\beta(A_n)}{\mu_\pi(A_n)} = \infty.$$

and there is possibly an exponential relationship of the kind exposed in proposition 3.

The expression  $A_2$  is one of the simplest  $\lambda$ -expressions for which  $\mu_\beta$  is strictly larger than  $\mu_\pi$ . Slightly modified versions of  $A_2$  are presented in [Lév88, Fie90, Lam90].

## Conclusion

- We have defined a notion of complexity with which to measure the efficiency of an implementation of the  $\lambda$ -calculus. In terms of this, the *complexity* of a functional program can be given a precise meaning.
- We show that all implementations must have at least linear complexity.
- We have devised a technique (as exemplified in proposition 2) for proving combinator based implementations inefficient.

## Open Problems

- Does there exist an optimal implementation? If yes, what is its complexity?
- What are the complexities of various well known implementations? (e.g. graph reduction.)
- Does there exist an implementation of polynomial ( $\nu^{O(1)}$ ) or linear ( $O(\nu)$ ) complexity?
- Is  $\mu_\beta$  bounded in terms of  $\mu_\pi$ ?

## Appendix

### Naturally Embedding Imperative Computations in the $\lambda$ -calculus whilst Conserving Complexity

We want to show that a conventional machine model can be simulated in the  $\lambda$ -calculus with only a constant amount of overhead.

#### Choice of Machine Model.

The most realistic choice would be a RAM. However, there are several variants differing in their basic instruction sets and their complexity measures. It seems that the idea of a “unit-cost” RAM is flawed from a theoretical standpoint, unless its arithmetic instruction set is restricted to contain only the successor function, and no shift operations are permitted (see [Sch80] for discussion and references). An alternative is the “logarithmic cost” RAM. However, this model also seems flawed in that it seems unable to read and store its input in linear time (see [Sch88] for details).

The Turing Machine (TM) is a more basic model, for which the choice of a basic instruction set does not influence the complexity measure. However, the TM is unrealistically slow, because its storage tapes only allow sequential access. Schönhages Storage Modification Machine overcomes this deficiency, but seems too powerful for our purposes. (See [HBTCS, ch.1] for a presentation and discussion of all these models).

We introduce a strengthened version of the TM. Our version uses trees rather than tapes as a storage medium. The tree TM has a complexity measure that is closely related to the RAM measures. Yet, the tree TM has the same simple instruction set as the conventional TM with tape storage.

#### Definition A.1

##### $k$ -tree Turing Machine

A  $k$ -tree Turing machine consists of two i/o tapes,  $k$  work-trees and a finite control. The input tape contains initially a string over  $\{0,1\}$  terminated by a ‘#’ and one head positioned at the left end of the tape. The head may read symbols and move to the right (and nothing else). The output tape is initially empty and has one head positioned at the left end of the tape. The head may

write symbols  $\{0,1,\#\}$  and move to the right, but nothing else. Both trees are initially empty, i.e. all cells contain the blank symbol, and each tree has one head that may both read and write in addition to moving stepwise in one of the directions  $\{1,2,3\}$ . In each step, a  $k$ -tree Turing machine reads the cell symbol under each of the  $(k+1)$  heads that are allowed to read, inspects the state, and based upon the transition function, the control changes the state, overwrites the symbol under each of the  $(k+1)$  heads that are allowed to write and moves some (or none) of the  $(k+2)$  heads in legal directions.

A  $k$ -tree Turing machine  $M$  is said to compute a function  $f : \{0,1\}^n \rightarrow \{0,1\}^m$  whenever the following holds:

For all  $(a_1, a_2, \dots, a_n)$  in  $\{0,1\}^n$ , we have  $f(a_1, a_2, \dots, a_n) = (b_1, b_2, \dots, b_m)$  if and only if  $M$  started on an input tape ' $a_1, a_2, \dots, a_n, \#$ ' stops with output tape ' $b_1, b_2, \dots, b_m, \#$ '.

The complexity  $T_M(n)$  of the machine  $M$  is the maximum number of steps taken before halting on any input of length  $n$ . The size of a  $k$ -tree Turing Machine with  $q$  states and  $s$  symbols is  $q \cdot s^k \cdot \log(q \cdot s^k)$ .

It should be clear to the reader that the complexity measure defined by the  $k$ -tree TM is very close to that defined by the various RAM models. More or less all the models discussed above define complexities that differ by at most a log factor. Informally, a  $k$ -tree TM may be used to simulate a RAM by using one of its trees to simulate the RAM's memory with binary addressing in the intuitive way. A second tree may be used to assist with bookkeeping and arithmetic instructions. Thus a similar complexity measure to the logarithmic cost RAM is obtained, but at much greater simplicity. The reasons for our choice of the  $k$ -tree TM will become apparent.

## Definition A.2

### Control and data structures encoded as $\lambda$ -expressions

We give specific names to some very useful  $\lambda$ -expressions:

- The truth-values:

$$\begin{aligned} \mathbf{t} &= (\lambda xy.x) \\ \mathbf{f} &= (\lambda xy.y) \end{aligned}$$

These are conveniently used for a conditional control expression

**if**  $c$  **then**  $L$  **else**  $N$  **fi**

to be coded as

$$(cLN)$$

We may also encode an iterative (recursive) control expression

$$Pl_0$$

where

$$PL = \mathbf{if} (cL) \mathbf{then} (P(pL)) \mathbf{else} (qL) \mathbf{fi}$$

using the least fixed point combinator  $Y$  from definition 1:

$$P = Y(\lambda f. \lambda L. (cL)(f(pL))(qL))$$

- The data structure *list*

$$\begin{aligned} [ ] &= (\lambda x. I) \\ [M_0, M_1, \dots, M_n] &= (\lambda x. x M_0 [M_1, \dots, M_n]) \end{aligned}$$

with the indexing functions

$$\begin{aligned} tl_n &= (\lambda x. x \mathbf{ff} \dots \mathbf{f}) \\ hd_n &= (\lambda x. (tl_n x) \mathbf{t}) \end{aligned}$$

where the defining expression for  $tl_n$  contains  $n$  copies of  $\mathbf{f}$  and

$$\begin{aligned} tl_k[M_0, M_1, \dots, M_n] &= [M_k, \dots, M_n] \\ hd_k[M_0, M_1, \dots, M_n] &= [M_k] \end{aligned}$$

- Unary integers.

$$u_n = hd_n$$

### Definition A.3

#### $\lambda$ -programs

Let  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  be a partial function on Boolean strings. A  $\lambda$ -expression  $N$  computes  $f$  when

1. if  $f(b_1, \dots, b_n) = (c_1, \dots, c_m)$  then  $L = (N[u_{b_1}, \dots, u_{b_n}, u_2])$  has the normal form  $[u_{c_1}, \dots, u_{c_m}, u_2]$ .
2. if  $f(b_1, \dots, b_n)$  is undefined then  $L$  has no normal form.

### Proposition A.4

#### Efficient simulation of tree Turing Machines in the $\lambda$ -calculus

For every 2-tree TM  $M$  there is a  $\lambda$ -expression  $N$  and a constant  $c$  such that  $N$  and  $M$  compute identical functions and if  $M$  on input  $(b_1, \dots, b_n)$  halts after  $t$  steps then the  $\lambda$ -expression  $L = (N[u_{b_1}, \dots, u_{b_n}, u_2])$  satisfies that  $\mu_\pi(L) = \mu_\beta(L) = l_{normal}(L) \leq c \cdot t$ .

#### Proof

For a given 2-tree Turing Machine  $M$  we construct a  $\lambda$ -program as follows.

Both symbols and states are represented by unary numbers such that for *symbols*

1.  $u_0$  and  $u_1$  denote the bit-symbols 0 and 1.
2.  $u_2$  denote the tape end-marker  $\#$ .
3.  $u_3$  denotes the blank symbol.

and for *states*

1.  $u_0$  denotes the initial state.
2.  $u_1$  denotes the unique halt state.

An instantaneous description of the 2-tree Turing Machine is kept as a list of five elements

$$\mathbf{id} = [\mathbf{state}, \mathbf{tree}_1, \mathbf{tree}_2, \mathbf{tape}_i, \mathbf{tape}_o]$$

One may regard a tree as having a root at the head position and being directed away from this root, in which case each node in the tree has two outgoing edges and one incoming edge except for the root that has three outgoing edges and no incoming edges. Every subtree may be represented as the symbol at the subtree-root combined with a subtree for each edge that goes out of the root. For simplicity, we take a subtree for each incident edge:

$$\mathbf{tree} = [\mathbf{t}, \mathbf{symbol}, \mathbf{tree}_1, \mathbf{tree}_2, \mathbf{tree}_3]$$

where in the case of an incoming edge, the corresponding tree is empty, represented by

$$tree_e = [\mathbf{f}, u_3]$$

The empty tree is also used as a representation of unexplored subtrees that have blank symbols at all nodes. The truth-value occurring as the first element in the list representation of a tree is used to distinguish the empty tree.

A tape is represented by a recursive list

$$\mathbf{tape} = [\mathbf{symbol}, \mathbf{tape}]$$

that may be empty

$$tape_e = [u_2]$$

To see how a single step of a computation on the 2-tree Turing Machine may be simulated, we look at a representative example. Assume that in the current  $\mathbf{id}$ , we must make the following changes

1. The finite state control goes into  $state'$ .
2.  $symbol'_1$  is written under the head of  $tree_1$ , which stays.
3.  $symbol'_2$  is written under the head of  $tree_2$ , which moves one step in direction 1.
4. The head of the input tape is moved one step (necessarily to the right).

5.  $symbol'$  is written on the output tape (and the head is moved one step to the right).

The following  $\lambda$ -expression will applied to an **id** reduce to a modified **id** as prescribed above.

$$\begin{aligned}
 & (\lambda x. [ \text{state}', \\
 & \quad [ \mathbf{t}, symbol'_1, tl_2(hd_1 x)], \\
 & \quad [ \mathbf{t}, \\
 & \quad \quad hd_1(hd_2(hd_2 x)), \\
 & \quad \quad [ \mathbf{t}, symbol'_2, tree_e, tl_3(hd_2 x)], \\
 & \quad \quad hd_0(hd_2(hd_2 x)) \ tl_3(hd_2(hd_2 x)) \ [tree_e, tree_e] \\
 & \quad ] \\
 & \quad tl_1(hd_3 x), \\
 & \quad [symbol', hd_4 x] \\
 & ] )
 \end{aligned}$$

The most complicated part of the above expression is the modification of  $tree_2$ . It is here necessary to provide two alternative sub-actions according to whether the subtree in direction 1 has been visited before or not. The choice is made by the truth-valued expression  $hd_0(hd_2(hd_2 x))$ .

The state and symbols under the heads in the present **id** determines which action to take. All possible actions are therefore organised in a 4-dimensional array *transitiontable* that is indexed by

1. state
2. symbol under head of  $tree_1$
3. symbol under head of  $tree_2$
4. symbol under input head

The simulation of one single transition step may thus be done by the following  $\lambda$ -expression that updates its argument (an **id**) appropriately:

$$F = (\lambda x. hd_0(tl_3 x)(hd_1(tl_2 x)(hd_1(tl_1 x)(hd_0 x \text{ transitiontable}))))x)$$

We are now in a position to describe the whole simulation. For control purposes we define a constant expression

$$halt = [\mathbf{f}, \mathbf{t}, \mathbf{f}, \mathbf{f}, \dots, \mathbf{f}]$$

with the property that ' $u_i \text{ halt}$ ' reduces to  $\mathbf{t}$  precisely when  $u_i$  denotes the halt state (The size of the expression for halt is proportional to the number of states in the machine  $M$ ). Assume the  $\lambda$ -program  $G$  on argument  $id$  reduces to the output tape eventually computed if  $id$  leads to a halting configuration, and otherwise  $G$  does not reduce to a normal form. This  $G$  may be characterised recursively:  $G = (\lambda x. hd_0 x \text{ halt}(hd_4 x)(G(F x)))$ , i.e. if the present configuration contains a halt state then the result consists in the present output tape ( $hd_4 x$ ), otherwise apply  $G$  to the result of yet another iteration,  $G(F x)$ . Hence  $G$  may be defined by

$$G = Y(\lambda g. \lambda x. hd_0 x \text{ halt}(hd_4 x)(g(F x))).$$

$G$  gives the output tape in reverse order, and then we need a list reversal function. Define first the constant expression

$$\text{end} = [\mathbf{f}, \mathbf{f}, \mathbf{t}, \mathbf{f}, \mathbf{f}, \dots, \mathbf{f}]$$

with the property that ' $u_i \text{ end}$ ' reduces to  $\mathbf{t}$  precisely when  $u_i$  denotes the tape end marker  $\#$ . The reversal function is

$$\text{Rev} = (\lambda z. Y(\lambda h. \lambda x. \lambda y. hd_0 y \text{ end } x (h[hd_0 y, x](tl_1 y))) \ [u_2] \ z)$$

i.e.  $\text{Rev}[u_{i_1}, u_{i_2}, \dots, u_{i_n}, u_2]$  reduces to  $[u_{i_n}, u_{i_{n-1}}, \dots, u_{i_1}, u_2]$  where  $i_j \neq 2$ .

Define

$$id_0 = (\lambda x. [u_0, [\mathbf{t}, u_3, \text{tree}_e, \text{tree}_e, \text{tree}_e], [\mathbf{t}, u_3, \text{tree}_e, \text{tree}_e, \text{tree}_e], x, [u_2]])$$

which applied to an input tape  $[u_{b_0}, \dots, u_{b_n}, u_2]$  reduce to the initial  $\mathbf{id}$  of the 2-tree Turing Machine. Finally define the  $\lambda$ -program

$$N = (\lambda x. \text{Rev}(G(id_0 x)))$$

that computes the same function as the 2-tree Turing Machine  $M$  does.

It should be clear from the construction that the number of  $\beta$ -reduction steps needed for the simulation of a single TM-step is bounded by a constant. We find also that regular  $\beta$ -reduction in any order is as efficient as normal order parallel reduction, essentially because a list indexing function forces an order of reduction by creating only one redex at a given time. Where more than one indexing function is enabled the corresponding reduction steps are independent and may be done in any order. Hence  $\mu_\pi(L) = \mu_\beta(L) = l_{\text{normal}}(L)$ .

QED.

## References

- [Bar81] Barendregt, H. P., *The Lambda Calculus. Its Syntax and Semantics*. North Holland, 1981.
- [Bur82] Burton, F. W., A Linear Space Translation of Functional Programs to Turner Combinators. *Information Processing Letters*, 14 (1982), pp. 201-204.
- [deB72] de Bruijn, N. G., Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation. *Indag Math*, 34 (1972), pp. 381-392.
- [Chu32] Church, A., A Set of Postulates for the Foundation of Logic. *Annals of Math.* 33, 2nd series (1932), pp. 346-366.
- [CuFe58] Curry, H. B. and Feys, R. *Combinatory Logic, Vol. 1*. North Holland, 1958.
- [Fie90] Field, J., On Laziness and Optimality in Lambda Interpreters: Tools for Specification and Analysis. In [PoPL90], pp. 1-15.
- [HBTCS] *the Handbook of Theoretical Computer Science, vol A* (ed. J. van Leeuwen). Elsevier, Amsterdam, 1990.
- [Hug82] Hughes, R. J. M., Super Combinators: A New Implementation Method for Applicative Languages. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, pp.1-10.
- [Hug84] Hughes, R. J. M., *The Design and Implementation of Programming Languages*. Ph.D. Thesis, Oxford University, 1984. (PRG-40)
- [KeSl87] Kennaway, J. R. and Sleep, M. R., Variable Abstraction in  $O(n \log(n))$  Space. *Information Processing Letters*, 24 (1987), pp. 343-349.
- [Klo80] Klop, J. W., *Combinatory Reduction Systems*. Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980.

- [Lam90] Lamping, J., An Algorithm for Optimal Lambda Calculus Reduction. In [PoPL90], pp. 16-30.
- [Lév80] Lévy, Jean-Jacques, Optimal Reductions in the Lambda-Calculus. In Seldon, J. P. and Hindley, J. R. (editors), *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, 1980, pp. 159-191.
- [Lév88] Lévy, Jean-Jacques, Sharing in the Evaluation of Lambda Expressions. In Fuchi, K. and Kott, L. (editors), *Programming of Future Generation Computers II*, North Holland, 1988, pp. 183-189.
- [Nos85] Noshita, K., Translation of Turner Combinators in  $O(n \log(n))$  Space. *Information Processing Letters*, 20 (1985), pp. 71-74.
- [PoPL90] Proceedings of Seventeenth Annual ACM Symposium on Principles of Programming Languages. ACM, New York, 1990.
- [Rév88] Révész, G. E., *Lambda-Calculus, Combinators and Functional Programming*. Cambridge University Press, 1988.
- [Ros84] Rosser, J. B., Highlights of the History of the Lambda-Calculus. *Annals of the History of Computing* 6 (1984), pp. 337-349.
- [Sch24] Schönfinkel, M., Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92 (1924), pp. 305-316.
- [Sch80] Schönhage, A., Storage Modification Machines. *SIAM Journal on Computing* 9 (1980), pp. 490-508.
- [Sch88] Schönhage, A., A Nonlinear Lower Bound for Random-Access Machines under Logarithmic Cost. *Journal of the ACM* 35 (1988), pp. 748-754.
- [Sta82] Staples, J., Two-Level Expression Representation for Faster Evaluation. In Ehrig, H., Nagl, M. and Rozenberg, G. (editors), *Proceedings from 2nd International workshop "Graph Grammars and their Application to Computer Science"*. Springer Verlag (LNCS 153), 1983, pp. 392-404.

- [Tur79a] Turner, D. A., Another Algorithm for Bracket Abstraction. *The Journal of Symbolic Logic* 44 (1979), pp. 267-270.
- [Tur79b] Turner, D. A., New Implementation Techniques for Applicative Languages. *Software: Practice & Experience*. 9 (1979), pp. 31-49.