Lambda Calculi and Linear Speedups

David Sands, Jörgen Gustavsson, and Andrew Moran[⋆]

Department of Computing Science,
Chalmers University of Technology and Göteborg University, Sweden.
`www.cs.chalmers.se`

Abstract. The equational theories at the core of most functional programming are variations on the standard lambda calculus. The best-known of these is the call-by-value lambda calculus whose core is the value-beta computation rule $(\lambda x.M)\,V \to M[{}^{V}\!/_{x}]$ where V is restricted to be a *value* rather than an arbitrary term.
This paper investigates the transformational power of this core theory of functional programming. The main result is that the equational theory of the call-by-value lambda calculus cannot speed up (or slow down) programs *by more than a constant factor*. The corresponding result also holds for call-by-need but we show that it does not hold for call-by-name: there are programs for which a single beta reduction can change the program's asymptotic complexity.

1 Introduction

This paper concerns the transformational power of the core theory of functional programming. It studies some computational properties of call-by-value lambda calculus, extended with constants; the prototypical call-by-value functional language. We show that this equational theory has very limited expressive power when it comes to program optimisation: equivalent programs always have the same complexity. To develop this result we must first formulate the question precisely. What does it mean for lambda terms to have the same complexity? What is a reasonable measure of cost? First we review the role of lambda calculi in a programming language setting.

1.1 Lambda Calculi

Lambda-calculus provides a foundation for a large part of practical programming. The "classic" lambda calculus is built from terms (variables, abstraction and application), a reduction relation (built from beta and eta). The calculus itself is an *equational theory* built from reduction by compatible, transitive and symmetric closure of the reduction relation.

Although many (functional) programming languages are based on lambda calculus (Lisp, Scheme, ML, Haskell), the correspondence between the classical

[⋆] Author's current address: Galois Connections, Oregon. `moran@galcon.com` theory and the programming languages is not particularly good, and this mismatch has given rise to a number of *applied* lambda calculi – calculi more finely tuned to the characteristics of programming languages.

Plotkin's criteria [Plo76] for the correspondence between a programming language and a reduction calculus:

1. Standard derivations are in agreement with the operational semantics of the programming language (abstract machine e.g. SECD)
2. Equations for the calculus are operationally sound – that is, they are contained in some suitable theory of observational equivalence.

Lambda calculus (and combinatory-logic) based programming languages are typically implemented using one of two reduction strategies: *call-by-value* – e.g. Scheme, various ML dialects, which dictates that arguments to functions are evaluated before the call is made, and *call-by-need*, exemplified by implementations of Clean and Haskell, in which an argument in any given application is evaluated only when it is needed – and in that case it is not evaluated more than once.

Accepting Plotkin's criteria, the standard lambda calculus is not a good fit for lambda-calculus-based programming languages, since it is

– not operationally sound for call-by-value languages, since beta reduction is not sound for call-by-value, and
– not in agreement with standard derivations for call-by-need languages, since unlike call-by-need computation, beta reduction entails that arguments in any given call might be evaluated repeatedly.

As a consequence a number of "applied lambda calculi" have been studied, most notably:

– λ_v calculus [Plo76] and its conservative extensions to handle control and state [FH92]
– λ_{need} [AFM+95] – and later refinements [AB97]

The main result of this article is that the call-by-value lambda calculus cannot speed up (or slow down) programs *by more than a constant factor*. That is to say, the equational theory, although sufficient to support computation, can never yield a superlinear speedup (or slowdown). The corresponding result also holds for the lesser-known call-by-need calculus – a result which was outlined in [MS99].

In contrast, we also show that the result does *not* hold for the standard lambda calculus (based on full beta reduction) and the corresponding call-by-*name* computational model. In the case of full beta reduction there are a number of subtleties – not least of which is finding a machine model which actually conforms to the cost model of beta reduction.

1.2 Related Work

Much of the work on the border between programming languages and complexity has focused on characterising the computational power of various restricted programming languages. The present work relates most closely to characterisations of the range of transformations that are possible in particular program transformation systems. A few examples of negative results exist. The closest is perhaps Andersen and Gomard's [AG92, JGS93]. They considered a simple form of partial evaluation of flow chart programs and showed that superlinear speedups were not possible. Amtoft suggests that even expressive systems such as unfold-fold, in a particular restricted setting, can give only constant-factor speedups. Incompleteness results for unfold-fold transformation include the fact that these transformations cannot change the *parallel complexity* of terms [Zhu94, BK83], where *parallel complexity* assumes a computational model in which all recursions at the same nesting level are evaluated in parallel. Our earlier work [San96] on the use of improvement for establishing the *extensional* correctness of memoization-based program transformations showed that recursion-based *deforestation* in a call-by-name setting cannot speed up programs by more than a constant factor.

The result in this paper was first proved for call-by-need lambda calculus in [MS99], and the key development via an asymptotic version of *improvement theory*, was also introduced there.

1.3 Overview

The remainder of the paper is organised as follows:

Section 2 formalises what it means for one program to be asymptotically least as fast as another, beginning with a traditional definition, and strengthening it until it is adequate to handle a higher-order language. **Section** 3 introduces the call-by-value lambda-calculus and **Section** 4 discuss what is a reasonable model of computation cost for the calculus. **Section** 5 establishes the main result via a notion of *strong improvement.* **Section** 6 considers call-by-name computation and show that the result fails there. **Section** 7 concludes.

2 Relative Complexity

The main result which we prove in this article is that:

> If $M = N$ is provable in the call-by-value lambda calculus λ_v then M and N have the same asymptotic complexity with respect to the call-by-value computational model.

In this section we make precise the notion of "having the same asymptotic complexity" when M and N are (possibly open) lambda terms.

2.1 Improvement

We will characterise the notion of one program having the same complexity as another in terms of an *improvement* preorder. Let us begin with a "traditional" view of when one program is better than another. Firstly we will assume that we are comparing extensionally equivalent programs. Consider first the simpler case of two programs P and Q which compute some function $f \in D \to E$. Let p and $q \in \mathbb{N} \to \mathbb{N}$ denote the time functions for P and Q respectively, where $p(n)$ is the worst-case number of computation steps required to compute P on an input of size n.

In traditional algorithm analysis style one would say that P is asymptotically at least as fast as Q (or equivalently, P is no more than a constant factor slower than Q) if

$$p(n) \in \mathcal{O}(q(n)).$$

More precisely, if there exit constants a and n_0 such that

$$\forall n > n_0 . p(n) \leq a \cdot q(n). \tag{2.1}$$

We will work with a definition which is both

– stronger – yielding a stronger main result, and
– more general – since we intend to deal with lambda terms and program fragments rather than whole programs computing over "simple" domains.

Strengthening Let us first deal with the strengthening of the condition. The above definition is based on the notion of the *size* of a given input. As well as being difficult to generalise to a lambda calculus, it also makes for a weaker definition, since the time functions give the worst case running time for each input size. For example, suppose we have a quicksort algorithm Q which is $\mathcal{O}(n^2)$ in the worst case (e.g., for fully sorted lists), but performs much better for a large part of the input domain. According to the above definition, a sorting algorithm B implementing bubble sort, which exhibits quadratic behaviour in *every* case, would be considered asymptotically equivalent to Q.

By using a stronger notion of "asymptotic equivalence" of programs our main result will be stronger – enabling us to conclude that bubble sort could never be transformed into quicksort using the calculus. With this in mind, an obvious strengthening of the notion of asymptotic speedup – without resorting to average-case complexity – is to quantify over all inputs rather than just the worst-case for each input size. Firstly, we note that 2.1 can be written equivalently as

$$\exists a, b \in \mathbb{N} . \forall n . p(n) \leq a \cdot q(n) + b \tag{2.2}$$

The equivalence can be seen by taking $b = \max\{p(m) \mid m \leq n_0\}$. This is a useful form since it eliminates the use of the inequality relation – a prerequisite for generalising to inputs which do not have a natural "size" measure. To make the desired strengthening, let $\mathrm{time}_P(d)$ denote the running time of P on input

d, and similarly for Q. The strengthening of the definition is to say that Q is *improved by* P iff

$$\exists a, b \in \mathbb{N}.\, \forall d \in D.\, \mathrm{time}_P(d) \leq a \cdot \mathrm{time}_Q(d) + b$$

We would then say that P has the same asymptotic complexity as Q if P is improved by Q, and Q is improved by P.

Generalisation Having strengthened our notion of asymptotic improvement, we now seek a *generalisation* which permits us to deal with not just whole programs but also program fragments. The generalisation is to replace quantification over program inputs by quantification over the *contexts* in which the program fragment can occur. A context (for a given programming notation), written $C[\cdot]$, is a program containing a hole $[\cdot]$; $C[M]$ denotes the result of replacing the hole by program fragment M – possibly capturing free variables that occur in M in the process.

We identify the concept of *program* with a closed term.

Definition 1 (Asymptotic Improvement). *Given a programming language equipped with*

- *a notion of operational equivalence between terms, and*
- *an operational model (abstract machine) which defines a partial function* $time_P$*, the time (number of steps) to evaluate program* P *(i.e.* $time_P$ *is well defined whenever* P *terminates)*

define the following relation on terms:

Let M *and* N *be operationally equivalent terms.* N *is no more than a constant factor slower than* M *if there exist integers* a *and* b *such that for all contexts* C *such that* $C[M]$ *and* $C[N]$ *are closed and terminating,*

$$time_{C[N]} \leq a \cdot time_{C[M]} + b$$

We alternatively say that M is asymptotically improved by N*, and write* $M \mathrel{\underset{\approx}{\triangleright}} N$*. If both* $M \mathrel{\underset{\approx}{\triangleright}} N$ *and* $N \mathrel{\underset{\approx}{\triangleright}} M$ *then we write* $M \mathrel{\underset{\approx}{\triangleleft\!\triangleright}} N$*, and say that* M *and* N *are* asymptotically cost equivalent*.*

A straightforward but important consequence of this definition is that $\mathrel{\underset{\approx}{\triangleright}}$ is a *congruence* relation: it is transitive, reflexive, symmetric and preserved by contexts, i.e, $M \mathrel{\underset{\approx}{\triangleright}} N$ implies that $\mathbb{C}[M] \mathrel{\underset{\approx}{\triangleright}} \mathbb{C}[N]$.

3 The $\boldsymbol{\lambda_v}$ Calculus

We will work with a syntax containing just lambda calculus and some arbitrary constants. This is essentially just Landin's ISWIM [Lan66, Plo76]. We follow Felleisen and Hieb's presentation [FH92] fairly closely, which in turn summarises Plotkin's introduction of the λ_v calculus [Plo76].

The language Λ consists of terms which are either *values* or *applications*:

$$\begin{array}{ll} \text{Terms } L, M, N & ::= V \mid M\,N \\ \text{Values } V, W & ::= b \mid f \mid x \mid \lambda x.M \end{array}$$

Values include variables and two kinds of constants, the basic constants ($b \in$ BConsts) and functional constants ($f \in$ FConsts). The constants are intended to include basic types like integers and booleans, and functional types include arithmetical and logical operations on these types. The exact meaning of the constants is given by a partial function:

$$\delta : \text{FConsts} \times \text{BConsts} \rightarrow \text{Values}^0$$

where Values^0 denotes the set of closed values (and similarly Λ^0 the set of closed terms). We impose the restriction that there is only a finite set of constants so that operations on constants can be implemented in constant time. We adopt the usual conventions of identifying terms which only differ in the names of their bound variables.

Operational Semantics Computation is built up from the following basic reduction rules:

$$\begin{array}{rll} f\,b \mapsto & \delta(f, a) & (\delta) \\ (\lambda x.M)\,V \mapsto & M[{}^{V}\!/_{x}] & (\beta_v) \end{array}$$

The operational semantics is a deterministic restriction of the application of the basic reduction rules which follows the "call-by-value" discipline of ensuring that arguments are evaluated before the function application can be evaluated. The computational order can be specified by only permitting reductions to take place in an *evaluation context*. Evaluation contexts are term contexts where the hole appears in the unique position we may perform a reduction:

$$\text{Evaluation Contexts } \mathbb{E} ::= [\cdot] \mid \mathbb{E}\,M \mid V\,\mathbb{E}$$

The small-step operational semantics is then given by the rule:

$$\mathbb{E}[M] \longmapsto \mathbb{E}[N] \text{ iff } M \mapsto N$$

It is easy to see that this one-step computation relation is deterministic. We write $M \longmapsto^n N$ to mean that M computes in $n \geq 0$ steps to N. Complete evaluation is defined by the repeated computation until a value is obtained:

Definition 2.

- *$M{\Downarrow}^n V$ iff $M \longmapsto^n V$.*
- *$M{\Downarrow}^n$ iff there exists V such that $M{\Downarrow}^n V$*
- *$M{\Downarrow}^{\leq m}$ iff there exists n such that $M{\Downarrow}^n$ and $n \leq m$.*

When $M{\Downarrow}^n$ we say that M converges *in n steps.*

The λ_v Calculus The λ_v calculus is the least equivalence relation ($=_v$) containing the basic reduction relation ($\mapsto$) and which is closed under all contexts:

$$M =_v N \implies \mathbb{C}[M] =_v \mathbb{C}[N]$$

where $\mathbb{C}$ denotes an arbitrary term context, and $\mathbb{C}[M]$ denotes the textual replacement of the (zero or more) holes in $\mathbb{C}$ by the term M. I.e., $=_v$ is the transitive, reflexive, symmetric and compatible closure of $\mapsto$. When $M =_v N$ it is also standard to write $\lambda_v \vdash M = N$.

4 Cost Models for the λ_v Calculus

It is tempting to take the counting of reduction steps as the natural model of computation cost for the λ_v calculus, since it is a simple and high level definition. But is this a reasonable choice? It is a crucial question since if we make an erroneous choice then our results would not say anything about actual implementations and they would be of little value.

We believe that a reasonable requirement of a cost model is that it should be implementable within a program size dependent constant factor. I.e., there should exist a linear function h such that, for every program M, if the cost of evaluating M in the model is n then the actual cost is within $|M| \cdot h(n)$ where $|M|$ denotes the size of M.

One could alternatively insist that the constant to be independent of the program, i.e., that there should exist a linear function h such that for every program M, if the cost of evaluating M in the model is n then the actual cost is within $h(n)$. That is a much stronger requirement, used in e.g. Jones "constant-factor-time hierarchy" work [Jon93, Ros98]. We believe that the results in this paper could be extended to such models, although we have not done so.

In the remainder of this section we will argue that to count reduction steps is a valid cost model for the λ_v calculus in former sense. Although it may seem intuitively obvious, it is a subtle matter as we will see when we turn to the call-by-name case later in this paper.

4.1 An Abstract Machine

Here we present a heap based abstract machine and argue informally that it can be implemented (e.g., in the RAM model) within a constant factor linearly dependent on the program size. Later we will relate the number of abstract machine steps to the model based on counting reduction steps.

For the purpose of the abstract machine we extend the syntax of the language with heap variables, we use p to range over those and we write $M[{}^{p}\!/\!_{x}]$ for substitution of a heap variable p for term variable x. We let Γ and S range over heaps and stacks respectively. Heaps are mappings from heap variables to values. We will write $\Gamma\{p = V\}$ for the extension of Γ with a binding for p to

V. Stacks are a sequence of "shallow" evaluation contexts of the form $[\cdot]\,N$ or $V\,[\cdot]$, given by:

$$\text{Stacks } S := \epsilon \mid [\cdot]\,N : S \mid V\,[\cdot] : S$$

where ϵ denotes the empty stack. Configurations are triples of the form $\langle\, \Gamma,\ M,\ S\,\rangle$ and we will refer to the second component M as the *control* of the configuration. The transitions of the abstract machine are given by the following rules.

$$\begin{array}{llrlrrl}
\langle\, \Gamma, & M\,N, & S\rangle \rightsquigarrow \langle\, \Gamma, & M, & [\cdot]\,N : S\rangle & \\
\langle\, \Gamma, & f, & [\cdot]\,N : S\rangle \rightsquigarrow \langle\, \Gamma, & N, & f\,[\cdot] : S\rangle & \\
\langle \Gamma\{p{=}V\}, & p, & S\rangle \rightsquigarrow \langle\, \Gamma\{p{=}V\}, & V, & S\rangle & \\
\langle\, \Gamma, & b, & f\,[\cdot] : S\rangle \rightsquigarrow \langle\, \Gamma, & \delta(f,a), & S\rangle & (\delta) \\
\langle\, \Gamma, & \lambda x.M, & [\cdot]\,N : S\rangle \rightsquigarrow \langle\, \Gamma, & N, & (\lambda x.M)\,[\cdot] : S\rangle & \\
\langle\, \Gamma, & V, & (\lambda x.M)\,[\cdot] : S\rangle \rightsquigarrow \langle\, \Gamma\{p{=}V\}, & M[{}^{p}\!/\!{}_{x}], & S\rangle & (\beta)
\end{array}$$

A crucial property of the abstract machine is that all terms in configurations originate from subterms in the original program, or from subterms of the values returned by $\delta(f,b)$. More precisely, for each term M in a configuration there exist a substitution σ mapping variables to heap variables and a term N which is a subterm of the original program or a subterm of the values in $\mathit{range}(\delta)$ such that $M \equiv N\sigma$. The property is sometimes called *semi compositionality* [Jon96]. The property is ensured because terms are never substituted for variables, thus the terms in the abstract machine states are all subterms (modulo renaming of variables for heap variables) of the original program. This is not the case for the reduction semantics, where terms may grow arbitrarily large as the computation proceeds.

Thanks to semi compositionality it is easy to see that each step can be implemented in time proportional to the maximum of the size of the original program and the size of the values in $\mathit{range}(\delta)$[1]. This is the only property we require of the machine steps.

4.2 Relating the Cost Model to the Abstract Machine

In the remainder of this section we will prove that the number of abstract machine steps required to evaluate a program is within a *program independent* constant factor of the number of reduction steps, as stated in the following lemma.

Lemma 1. *If $M{\Downarrow}^n$ then there exist Γ and V such that*

$$\langle\, \emptyset,\ M,\ \epsilon\,\rangle \rightsquigarrow^{\leq 6n} \langle\, \Gamma,\ V,\ \epsilon\,\rangle$$

Together with our informal argument that the abstract machine can be implemented within a program dependent constant factor this shows that to count reduction steps is a valid cost model of the λ_v-calculus.

[1] This set is finite and independent of the program in question.

We will prove Lemma 1 in two steps. In the first we show that the number of reduction steps in the reduction semantics is the same as the number of abstract machine transitions which use one of the rules (β) or (δ).

Lemma 2. *If $M{\Downarrow}^n$ then there exist Γ and V such that*

$$\langle\, \emptyset,\ M,\ \epsilon \,\rangle \leadsto^* \langle\, \Gamma,\ V,\ \epsilon \,\rangle$$

using exactly n applications of (β) and (δ)

The proof of this lemma is by standard technique relating small-step semantics and abstract machines and is omitted. Next we claim that the total number of abstract machine transitions is at most six times the number of (β) and (δ) transitions.

Lemma 3. *If $\langle\, \emptyset,\ M,\ \epsilon \,\rangle \leadsto^m \langle\, \Gamma,\ V,\ \epsilon \,\rangle$ using n applications of (β) and (δ) then $m \leq 6n$.*

Taken together with Lemma 2 this immediately implies Lemma 1.

In the remainder of this section we prove Lemma 3. First we define a measure $\lceil \cdot \rceil$ on terms and stacks as follows.

$$\begin{aligned} \lceil M \rceil &= \begin{cases} 0 & \text{if } M = p \\ 1 & \text{otherwise} \end{cases} \\ \lceil \epsilon \rceil &= 0 \\ \lceil [\cdot]\, M : S \rceil &= 2 + \lceil S \rceil \\ \lceil V\, [\cdot] : S \rceil &= 4 + \lceil S \rceil \end{aligned}$$

With the help of the measure we can generalise Lemma 3 to configurations with a non empty heap and stack.

Lemma 4. *If $\langle\, \Gamma,\ M,\ S \,\rangle \leadsto^m \langle\, \Gamma',\ V,\ \epsilon \,\rangle$ using n applications of (β) and (δ) then $m \leq 6n - \lceil M \rceil - \lceil S \rceil + 1$.*

Proof. The proof of Lemma 4 is by induction over m.

case $m = 0$: In the base case we have that $n = 0$ and that $\langle\, \Gamma,\ M,\ S \,\rangle \equiv \langle\, \Gamma',\ V,\ \epsilon \,\rangle$. Thus

$$6n - \lceil M \rceil - \lceil S \rceil + 1 = 6n - \lceil V \rceil - \lceil \epsilon \rceil + 1 = 0 = m$$

as required.

case $m > 0$: We proceed by a case analysis on the abstract machine rule in question. We will only consider (β) and the rule for applications. The other cases follows similarly.

subcase (β): In this case $\langle \Gamma,\ M,\ S \rangle \equiv \langle \Gamma,\ V,\ (\lambda x.N)\,[\cdot] : T \rangle$ for some V, N and T and we know from the induction hypothesis that

$$6(n-1) - \lceil N[{}^{p}\!/_{x}] \rceil - \lceil T \rceil + 1 \geq m - 1$$

for some p. The required result then follows by the following calculation.

$$\begin{aligned} 6n - \lceil M \rceil - \lceil S \rceil + 1 &= 6n - \lceil V \rceil - \lceil (\lambda x.N)\,[\cdot] : T \rceil + 1 \\ &= 6n - 1 - 4 - \lceil T \rceil + 1 \\ &= 6(n-1) - \lceil N[{}^{p}\!/_{x}] \rceil - \lceil T \rceil + 1 + 1 + \lceil N[{}^{p}\!/_{x}] \rceil \\ &\geq m - 1 + 1 + \lceil N[{}^{p}\!/_{x}] \rceil \\ &\geq m \end{aligned}$$

subcase (application rule): In this case $\langle \Gamma,\ M,\ S \rangle \equiv \langle \Gamma,\ N\,L,\ S \rangle$ for some N and L and we know from the induction hypothesis that

$$6n - \lceil N \rceil - \lceil [\cdot]\,L : S \rceil + 1 \geq m - 1.$$

The required result then follows by the following calculation.

$$\begin{aligned} 6n - \lceil M \rceil - \lceil S \rceil + 1 &= 6n - \lceil N\,L \rceil - \lceil S \rceil + 1 \\ &= 6n - \lceil S \rceil \\ &= 6n - \lceil N \rceil - \lceil [\cdot]\,L : S \rceil + 1 + 1 + \lceil N \rceil \\ &\geq m - 1 + 1 + \lceil N \rceil \\ &\geq m \end{aligned}$$

□

5 Constant Factors

We instantiate the definition of asymptotic improvement with the call-by-value computation model above. I.e., we take

$$\text{time}_M = n \iff M {\Downarrow}^n.$$

In the remainder of this section we will demonstrate the proof of our claim that the call-by-value lambda calculus cannot change asymptotic complexity. In other words, we show that $=_v$ is contained in $\underset{\approx}{\Leftrightarrow}$.

5.1 Strong Improvement

The main vehicle of our proof is a much stronger improvement relation which doesn't even permit constant factors. The reason why we make use of this relation is that it is "better behaved" semantically speaking. In particular it possesses relatively straightforward proof methods such as a *context lemma.*

Definition 3 (Strong Improvement). *M is strongly improved by N iff for all contexts $\mathbb{C}$ such that $\mathbb{C}[M]$ and $\mathbb{C}[N]$ are closed*

$$\mathbb{C}[M]{\Downarrow}^{n} \Rightarrow \mathbb{C}[N]{\Downarrow}^{\leq n}.$$

Following [MS99], we write this relation as $M \underset{\sim}{\trianglerighteq} N$. If $M \underset{\sim}{\trianglerighteq} N$ and $N \underset{\sim}{\trianglerighteq} M$ we say that M and N are cost equivalent and we write it as $M \underset{\sim}{\triangleleft\!\triangleright} N$.

Cost equivalence is a very strong relation since it requires that the two program fragments use exactly the same number of computation steps in all possible program contexts. For example $(\lambda x.M)\, V$ is not cost equivalent to $M[{}^{V}\!/_{x}]$ because the former takes up one more step than the latter each time it is executed.

5.2 The Tick

We introduce a technical device (widely used in our earlier work with improvement) for syntactically representing "a single computation step" – a dummy cost called a *tick*. We will denote the addition of a dummy cost to M as ${}^{\checkmark}M$. It is not a language extension proper since "the tick" can be encoded in the language as

$${}^{\checkmark}M \stackrel{\text{def}}{=} (\lambda x.M)\,() \qquad \text{where } x \notin \mathsf{FV}\,(M)$$

where $()$ is an arbitrary basic constant[2].

We can now state the fact that a reduct is cost equivalent to its redex if we add a dummy cost to it.

Lemma 5.

- $f\, b \underset{\sim}{\triangleleft\!\triangleright} {}^{\checkmark}\delta(f, a)$
- $(\lambda x.M)\, V \underset{\sim}{\triangleleft\!\triangleright} {}^{\checkmark}M[{}^{V}\!/_{x}]$

Although this result is intuitively obvious, it is hard to prove it directly since it involves reasoning about computation in arbitrary program contexts. The complications in the proof can be packaged up in a *Context Lemma* [Mil77] which provides a convenient way to show strong improvements by investigating the behaviour of terms in evaluation contexts.

Lemma 6 (Context Lemma). *Let σ range over substitutions mapping variables to closed values. If for all $\mathbb{E}$ and σ, $\mathbb{E}[M\sigma]{\Downarrow}^{n} \Rightarrow \mathbb{E}[N\sigma]{\Downarrow}^{\leq n}$ then $M \underset{\sim}{\trianglerighteq} N$.*

The proof of the context lemma is a simple extension of an established technique and we omit it here. A detailed proof of the context lemma for strong improvement for call-by-need can be found in [MS99]. With the help of the Context Lemma the proof of Lemma 5 is immediate by the virtue of the fact that computation is defined as reduction in evaluation contexts.

The next step in our proof is a claim that a tick cannot change the asymptotic behaviour of a term.

[2] An alternative would be to define tick to be an identity function. But due to the call-by-value model, this would not give ${}^{\checkmark}M \longmapsto M$.

$$\text{Value}\frac{}{V \Downarrow^{0} V} \qquad (\delta)\frac{M \Downarrow^{n_0} f \qquad N \Downarrow^{n_1} b}{M\,N \Downarrow^{n_0+n_1+1} \delta(f,b)}$$

$$(\beta_v)\frac{M \Downarrow^{n_0} \lambda x.L \qquad N \Downarrow^{n_1} V \qquad L[{}^{V}\!/_{x}] \Downarrow^{n_2} W}{M\,N \Downarrow^{n_0+n_1+n_2+1} W}$$

Fig. 1. The big-step cost semantics

Lemma 7. ${}^{\checkmark}M \mathrel{\underset{\approx}{\lhd\!\rhd}} M$

The intuitive argument for why this holds is that the execution of $\mathbb{C}[{}^{\checkmark}M]$ differs from the execution of $\mathbb{C}[M]$ only in some interleaving steps due to the tick, and these steps are dominated by the other steps. This is because the tick cannot stack up syntacticly during the computation, and as a consequence there is always a bound to the number of consecutive tick steps, and each such group can be associated to a "proper" reduction.

5.3 The Main Result

We will soon turn our attention to a rigorous proof of Lemma 7 but let us first show how our main result follows from Lemma 5 and 7.

Theorem 1.

$$=_v \subseteq \mathrel{\underset{\approx}{\lhd\!\rhd}}.$$

Proof. Assume that $M \mapsto N$. We can then make the following calculation

$$M \mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{\checkmark}N \mathrel{\underset{\approx}{\lhd\!\rhd}} N$$

Since $\mathrel{\underset{\sim}{\lhd\!\rhd}}$ is contained in $\mathrel{\underset{\approx}{\lhd\!\rhd}}$ it follows by transitivity that $M \mathrel{\underset{\approx}{\lhd\!\rhd}} N$, so $\mapsto$ is contained in $\mathrel{\underset{\approx}{\lhd\!\rhd}}$. Since $\mathrel{\underset{\approx}{\lhd\!\rhd}}$ is a congruence, we have that $=_v$, the congruent closure of $\mapsto$, is also contained in $\mathrel{\underset{\approx}{\lhd\!\rhd}}$. □

In the remainder of this section we will prove Lemma 7. It turns out that the proof is more conveniently carried out with the big-step semantics provided in Figure 1, which is the standard call-by-value big-step semantics augmented with a cost measure. The proof that the cost measure in the big-step semantics is in agreement with the small-step semantics is a trivial extension to the standard proof relating big-step and small-step call-by-value semantics.

Recall that a key point in the informal argument of why Lemma 7 holds is that ticks cannot "stack up" on top of each other during computation. To make this into a rigorous argument we introduce the ternary relation $M \overset{i\checkmark}{\succ} N, i \in \mathbb{N}$ defined in Figure 2 which intuitively means that M can be transformed into N by removing blocks of up to i consective ticks. A key property of the relation is that it satisfies the following substitution lemma.

$$\frac{}{{}^{j\checkmark}f \overset{i\checkmark}{\succ} f}\; j \leq i \qquad \frac{}{{}^{j\checkmark}b \overset{i\checkmark}{\succ} b}\; j \leq i \qquad \frac{}{{}^{j\checkmark}x \overset{i\checkmark}{\succ} x}\; j \leq i$$

$$\frac{M \overset{i\checkmark}{\succ} N}{{}^{j\checkmark}\lambda x.M \overset{i\checkmark}{\succ} \lambda x.N}\; j \leq i \qquad \frac{M_0 \overset{i\checkmark}{\succ} N_0 \qquad M_1 \overset{i\checkmark}{\succ} N_1}{{}^{j\checkmark}(M_0\, M_1) \overset{i\checkmark}{\succ} (N_0\, N_1)}\; j \leq i$$

Fig. 2. The tick erasure relation

Lemma 8. *If $M \overset{i\checkmark}{\succ} N$ and $V \overset{i\checkmark}{\succ} W$ then $M[{}^{V}\!/_{x}] \overset{i\checkmark}{\succ} N[{}^{W}\!/_{x}]$*

The lemma is easily proven by, for example, an induction over the structure of N.

The next step is to show that $\overset{i\checkmark}{\succ}$ is preserved by computation, and at the same time we show that the cost of executing the tick-decorated term is within a constant factor of the cost of executing the term without the ticks.

Lemma 9. *If $M \overset{i\checkmark}{\succ} N$ and $N \Downarrow^{n} W$ then there exists V such that*

- *$M \Downarrow^{m} V$,*
- *$m \leq (3i+1)n + i$ and*
- *$V \overset{i\checkmark}{\succ} W$.*

Proof. The proof is by well-founded induction over n. We will only consider the case when $N \equiv N_0\, N_1$. Then the derivation of $N \Downarrow^{n} W$ must be of the form

$$(\beta_v)\frac{N_0 \Downarrow^{n_0} \lambda x.N_2 \qquad N_1 \Downarrow^{n_1} W' \qquad N_2[{}^{W'}\!/_{x}] \Downarrow^{n_2} W}{N_0\, N_1 \Downarrow^{n_0+n_1+n_2+1} W}$$

where $n = n_0 + n_1 + n_2 + 1$. From $M \overset{i\checkmark}{\succ} N$ we know that M must be of the form ${}^{j\checkmark}(M_0\, M_1)$, for some $j \leq i$ and that $M_0 \overset{i\checkmark}{\succ} N_0$ and $M_1 \overset{i\checkmark}{\succ} N_1$. Thus it follows by two applications of the induction hypothesis that

- $M_0 \Downarrow^{m_0} \lambda x.M_2$, for some $m_0 \leq (3i+1)n_0 + i$,
- $\lambda x.M_2 \overset{i\checkmark}{\succ} \lambda x.N_2$,
- $M_1 \Downarrow^{m_1} V'$ for some $m_1 \leq (3i+1)n_1 + i$, and
- $V' \overset{i\checkmark}{\succ} W'$.

From $\lambda x.M_2 \overset{i\checkmark}{\succ} \lambda x.N_2$ and $V' \overset{i\checkmark}{\succ} W'$ it follows by Lemma 8 that $M_2[{}^{V'}\!/_{x}] \overset{i\checkmark}{\succ} N_2[{}^{W'}\!/_{x}]$ so we can apply the induction hypothesis a third time which gives that

- $M_2[{}^{V'}\!/_{x}] \Downarrow^{m_2} V$ where $m_2 \leq (3i+1)n_2 + i$, and
- $V \overset{i\checkmark}{\succ} W$.

We can now construct a derivation of $M_0\,M_1 \Downarrow^{m_0+m_1+m_2+1} V$ as

$$\frac{M_0 \Downarrow^{m_0} \lambda x.M_2 \qquad M_1 \Downarrow^{m_1} V' \qquad M_2[{}^{V'}\!/_{x}] \Downarrow^{m_2} V}{M_0\,M_1 \Downarrow^{m_0+m_1+m_2+1} V}$$

which gives that ${}^{j\checkmark}(M_0\,M_1) \Downarrow^{m_0+m_1+m_2+1+j} V$. We complete the proof with a calculation which shows that $m_0+m_1+m_2+1+j \leq (3i+1)n+i$.

$$\begin{array}{rl}
 & m_0+m_1+m_2+1+j \\
\leq & ((3i+1)n_0+i)+((3i+1)n_1+i)+((3i+1)n_2+i)+1+i \\
= & (3i+1)(n_0+n_1+n_2+1)+i \\
= & (3i+1)n+i
\end{array}$$

□

Finally we prove Lemma 7, i.e., that ${}^{\checkmark}M \overset{\diamond}{\approx} M$

Proof. We will start with the straightforward proof that ${}^{\checkmark}M \overset{\triangleright}{\approx} M$ and do the proof of $M \overset{\triangleright}{\approx} {}^{\checkmark}M$ thereafter. This direction is intuitively obvious since we remove the cost due to the tick. It is also easy to prove because it follows directly from the Context Lemma that ${}^{\checkmark}M \overset{\triangleright}{\sim} M$ and thus ${}^{\checkmark}M \overset{\triangleright}{\approx} M$.

Let us turn to the other direction. Assume that $\mathbb{C}[M]$ is closed and that $\mathbb{C}[M]{\Downarrow^{n}}$. Clearly $\mathbb{C}[{}^{\checkmark}M] \overset{1\checkmark}{\succ} \mathbb{C}[M]$ so it follows by Lemma 9 that $\mathbb{C}[{}^{\checkmark}M]{\Downarrow^{\leq 4n+1}}$ as required. □

6 Call-by-Name

A natural question to ask is whether our result carries over to other programming languages and their respective calculi. In other words, are calculi which fulfil Plotkin's criteria with respect to their intended programming language limited to linear speed-ups (or slow-downs)?

It turns out that the answer to this question is no, not in general. Here we show, somewhat surprisingly, that the main result fails for call-by-name if we take number of reductions as the measure of computation cost: full beta conversion can lead to asymptotic improvements (and worsenings) for programs in a programming language with normal order reduction. We show this with an example program for which a single beta reduction achieves a superlinear speedup.

We also discuss whether the number of reductions is a reasonable cost measure by comparing it to "the natural" abstract machine. It turns out – another surprise – that to count reductions steps in the reduction semantics is *not* in agreement (within a program size dependent constant factor) with a naïve version of the abstract machine. However a small optimisation of the abstract machine achieves an efficient implementation of the reduction semantics and thus we can justify that to count reductions is a cost measure that is implementable.

6.1 The $\boldsymbol{\lambda}_{name}$ Calculus

We begin by introducing the syntax and operational semantics. We will work with the same syntax as for the call-by-value language. The calculus $=_{name}$ is the congruent closure of the basic reductions:

$$\begin{array}{rll} f\,b \mapsto_{name} & \delta(f,a) & \quad (\delta) \\ (\lambda x.M)\,N \mapsto_{name} & M[{}^{N}\!/_{x}] & \quad (\beta) \end{array}$$

To specify the computation order we define the following reduction contexts:

$$\text{Call-by-name Evaluation Contexts } \mathbb{E} ::= [\cdot] \mid \mathbb{E}\,M \mid f\,\mathbb{E}$$

Then normal order reduction is just:

$$\mathbb{E}[M] \longmapsto_{name} \mathbb{E}[N] \text{ iff } M \mapsto_{name} N.$$

6.2 The $\boldsymbol{\lambda}_{name}$ Calculus and Superlinear Speedups

Theorem 2. *Beta-reduction can yield a superlinear speedup with respect to call-by-name computation.*

We will sketch the proof of this surprising proposition. We show that the removal of *a single tick* can lead to a superlinear speedup. I.e.,

$$M \underset{\approx_k}{\not\trianglerighteq} {}^{\checkmark}M$$

Since ${}^{\checkmark}M =_{name} M$ this shows that we cannot have $=_{name} \subseteq \underset{\approx}{\lhd\!\rhd}_{name}$.

So, how can the cost of a single tick dominate the cost of all other steps in the computation? To construct an example where this happens, we consider the proof for call-by-value. The point at which the call-by-value proof fails in the call-by-name setting is the substitution lemma for $M \overset{i\checkmark}{\succ} N$. In a call-by-name setting we cannot restrict ourselves to substitutions of values for variables as in Lemma 8. Instead we would need that,

$$\text{If } M_0 \overset{i\checkmark}{\succ} N_0 \text{ and } M_1 \overset{i\checkmark}{\succ} N_1 \text{ then } M_0[{}^{M_1}\!/_{x}] \overset{i\checkmark}{\succ} N_0[{}^{N_1}\!/_{x}]$$

which clearly fails. For example ${}^{\checkmark}x \overset{1\checkmark}{\succ} x$ and ${}^{\checkmark}M \overset{1\checkmark}{\succ} M$ but ${}^{\checkmark\checkmark}M \overset{1\checkmark}{\succ} M$ is not true because two ticks are nested on top of each other. Thus, ticks can stack up on top of each other during normal order computation (consider for example $(\lambda x.{}^{\checkmark}x)\,({}^{\checkmark}M)$). We will use this when we construct our counterexample.

The idea of the counterexample is that the computation first builds up a term with n ticks nested on top of each other. The term is then passed to a function which uses its argument m times. Since we are using a call-by-name language the argument will be recomputed each time so the total time will be $\mathcal{O}(nm)$. Let $\mathbb{C}_{m,n}$ denote the family of contexts given by

$$
\begin{array}{l}
\mathsf{let}\ (\circ) = \lambda f.\lambda g.\lambda x.f\ (g\ x)\\
\mathsf{in\ let}\ \mathit{apply} = \lambda f.\lambda x.f\ ({}^{\checkmark} x)\\
\quad \mathsf{in}\ \underbrace{(\mathit{apply} \circ \cdots \circ \mathit{apply})}_{n}\ (\lambda x.\overbrace{x + x + \cdots + x + x}^{m})\ 1\\
\longmapsto^{*}_{name}\\
(\lambda f.\lambda x.f\ ({}^{\checkmark}x))\ ((\lambda f.\lambda x.f\ ({}^{\checkmark}x))\ (\ldots((\lambda f.\lambda x.f\ ({}^{\checkmark}x))\ (\lambda x.x + x + \cdots + x + x))\ldots))\ 1\\
\longmapsto_{name}\\
(\lambda x.(\lambda f.\lambda x.f\ ({}^{\checkmark}x))\ (\ldots((\lambda f.\lambda x.f\ ({}^{\checkmark}x))\ (\lambda x.x + x + \cdots + x + x))\ldots)\ ({}^{\checkmark}x))\ 1\\
\longmapsto_{name}\\
(\lambda f.\lambda x.f\ ({}^{\checkmark}x))\ (\ldots((\lambda f.\lambda x.f\ ({}^{\checkmark}x))\ (\lambda x.x + x + \cdots + x + x))\ldots)\ ({}^{\checkmark}1)\\
\longmapsto_{name}\\
(\lambda x.(\ldots((\lambda f.\lambda x.f\ ({}^{\checkmark}x))\ (\lambda x.x + x + \cdots + x + x))\ldots)\ ({}^{\checkmark}x))\ \ ({}^{\checkmark}1)\\
\longmapsto_{name}\\
(\ldots((\lambda f.\lambda x.f\ ({}^{\checkmark}x))\ (\lambda x.x + x + \cdots + x + x))\ldots)\ ({}^{\checkmark\checkmark}1)\\
\longmapsto_{name}\\
\ldots\\
\longmapsto_{name}\\
(\lambda f.\lambda x.f\ ({}^{\checkmark}x))\ (\lambda x.x + x + \cdots + x + x)\ ({}^{\cdots\checkmark\checkmark}1)\\
\longmapsto_{name}\\
(\lambda x.(\lambda x.x + x + \cdots + x + x)\ ({}^{\checkmark}x))\ \ ({}^{\cdots\checkmark\checkmark}1)\\
\longmapsto_{name}\\
(\lambda x.x + x + \cdots + x + x)\ ({}^{\checkmark\cdots\checkmark\checkmark}1)\\
\longmapsto_{name}\\
\overbrace{\underbrace{({}^{\checkmark\cdots\checkmark\checkmark}1)}_{n} + \underbrace{({}^{\checkmark\cdots\checkmark\checkmark}1)}_{n} + \cdots + \underbrace{({}^{\checkmark\cdots\checkmark\checkmark}1)}_{n} + \underbrace{({}^{\checkmark\cdots\checkmark\checkmark}1)}_{n}}^{m}
\end{array}
$$

Fig. 3. A transition sequence where a tick stacks up

$$
\begin{array}{l}
\mathsf{let}\ (\circ) = \lambda f.\lambda g.\lambda x.f\ (g\ x)\\
\mathsf{in\ let}\ \mathit{apply} = \lambda f.\lambda x.f\ [\cdot]\\
\quad \mathsf{in}\ \underbrace{(\mathit{apply} \circ \cdots \circ \mathit{apply})}_{n}\ (\lambda x.\overbrace{x + x + \cdots + x + x}^{m})\ 1
\end{array}
$$

where we have used $\mathsf{let}\ x = M\ \mathsf{in}\ N$ as a short hand for $(\lambda x.N)\,M$.

The terms which exhibit the "stacking" of tick are $\mathbb{C}_{m,n}[{}^{\checkmark}x]$. Note that this tick then occurs in the definition of *apply*. When the program is executed this single tick builds up as shown by the reduction sequence in Figure 3. The term with the ticks is then duplicated when passed to g and to compute the resulting sum takes $\mathcal{O}(nm)$ time, where n is the number of calls to *apply* in the original term and m is the number of occurrences of x in g. If the tick is removed from the definition of apply then the ticks cannot build up and the resulting program runs in $\mathcal{O}(n+m)$ time. Thus we have a family of contexts, namely $\mathbb{C}_{m,n}$ which illustrate that ${}^{\checkmark}x \gtrapprox x$ cannot hold, since we can make n and m sufficiently large to defeat any constants which attempt to bound the difference in cost.

6.3 An Abstract Machine

Here we discuss whether counting reduction steps it is a reasonable cost model for call-by-name. We are not interested in whether we can do *better* than call-by-name (call-by-need and more "optimal" forms of reduction can do this), but whether there is an implementation of the language such that the number of reduction steps is a good model.

We start by introducing "the natural" heap based abstract machine, obtained by a minimal modification of the call-by-value machine. It is very similar to the call-by-value machine but the heap is now a mapping from heap variables to terms rather than a mapping from heap variables to values. The transitions of the machine are:

$$\begin{array}{lrrlrrl}
\langle \Gamma, & M\,N, & S\rangle \leadsto_{name} & \langle \Gamma, & M, & [\cdot]\,N : S\rangle & \\
\langle \Gamma, & f, & [\cdot]\,N : S\rangle \leadsto_{name} & \langle \Gamma, & N, & f\,[\cdot] : S\rangle & \\
\langle \Gamma\{p{=}N\}, & p, & S\rangle \leadsto_{name} & \langle \Gamma\{p{=}N\}, & N, & S\rangle & \text{(lookup)} \\
\langle \Gamma, & b, & f\,[\cdot] : S\rangle \leadsto_{name} & \langle \Gamma, & \delta(f,a), & S\rangle & (\delta) \\
\langle \Gamma, & \lambda x.M, & [\cdot]\,N : S\rangle \leadsto_{name} & \langle \Gamma\{p{=}N\}, & M[{}^{p}\!/\!_{x}], & S\rangle & (\beta)
\end{array}$$

The machine is still semi-compositional and it is thus easy to argue that the individual steps are implementable within a program size dependent constant factor. But is it a good implementation of our cost measure to count reduction steps? The answer is no, since the abstract machine may use arbitrarily more steps:

Lemma 10. *For every linear function h there is a program M which requires n reductions and m abstract machine steps where $m > |M| \cdot h(n)$.*

The problem with the abstract machine is that it may create chains in the heap of the following form.

$$\Gamma\{p = p_1, p_1 = p_2, \ldots, p_{n-1} = p_n, p_n = M\}$$

To evaluate p means to follow the chain by doing n consecutive lookups. If p is evaluated repeatedly then these lookup steps may dominate all other steps. To see how this can happen consider the following family of programs (based on our earlier example):

$$\begin{array}{l}
\mathsf{let}\ (\circ) = \lambda f.\lambda g.\lambda x.f\ (g\ x) \\
\mathsf{in}\ \mathsf{let}\ \mathit{apply} = \lambda f.\lambda x.f\ x \\
\quad \mathsf{in}\ \underbrace{(\mathit{apply} \circ \cdots \circ \mathit{apply})}_{n}\ (\lambda x.\overbrace{x + x + \cdots + x + x}^{m})\ 1
\end{array}$$

When we evaluate the term in this prototype abstract machine it builds up a chain

$$\Gamma\{p = p_1, p_1 = p_2, \ldots, p_{n-1} = p_n, p_n = 1\}$$

$$
\begin{array}{l}
\mathsf{let}\ (\circ) = \lambda f.\lambda g.\lambda x.f\ (g\ x)\\
\mathsf{in\ let}\ \mathit{apply} = \lambda f.\lambda x.f\ x\\
\quad \mathsf{in}\ \underbrace{(\mathit{apply} \circ \cdots \circ \mathit{apply})}_{n}\ (\lambda x.\overbrace{x + x + \cdots + x + x}^{m})\ 1\\
\longmapsto^{*}_{name}\\
(\lambda f.\lambda x.f\,x)\,((\lambda f.\lambda x.f\,x)\,(\ldots((\lambda f.\lambda x.f\,x)\,(\lambda x.x + x + \cdots + x + x))\ldots))\,1\\
\longmapsto_{name}\\
(\lambda x.(\lambda f.\lambda x.f\,x)\,(\ldots((\lambda f.\lambda x.f\,x)\,(\lambda x.x + x + \cdots + x + x))\ldots)\,x)\,1\\
\longmapsto_{name}\\
(\lambda f.\lambda x.f\,x)\,(\ldots((\lambda f.\lambda x.f\,x)\,(\lambda x.x + x + \cdots + x + x))\ldots)\,1\\
\longmapsto_{name}\\
(\lambda x.(\ldots((\lambda f.\lambda x.f\,x)\,(\lambda x.x + x + \cdots + x + x))\ldots)\,x)\ 1\\
\longmapsto_{name}\\
(\ldots((\lambda f.\lambda x.f\,x)\,(\lambda x.x + x + \cdots + x + x))\ldots)\,1\\
\longmapsto_{name}\\
\ldots\\
\longmapsto_{name}\\
(\lambda f.\lambda x.f\,x)\,(\lambda x.x + x + \cdots + x + x)\,1\\
\longmapsto_{name}\\
(\lambda x.(\lambda x.x + x + \cdots + x + x)\,x)\ 1\\
\longmapsto_{name}\\
(\lambda x.x + x + \cdots + x + x)\,1\\
\longmapsto_{name}\\
\overbrace{1 + 1 + \cdots + 1 + 1}^{m}
\end{array}
$$

Fig. 4. A transition sequence

of length n; eventually p is substituted for x in the body of

$$\lambda x.\overbrace{x + x + \cdots + x + x}^{m}.$$

and the chain is then traversed once for every p in

$$\overbrace{p + p + \cdots + p + p}^{m}.$$

Thus it takes (at least) $\mathcal{O}(nm)$ steps to evaluate a program in this family with the abstract machine. However if we turn to the reduction semantics it uses only $\mathcal{O}(n + m)$ as we can see by the schematic reduction sequence in Figure 4. This family of programs is enough to show that for every linear function h there is a program M which requires n reductions and m abstract machine steps where $m > h(n)$. It is not enough to prove Lemma 10 since the size of the programs in the family grows linearly in n and m. However it is possible to construct a family of terms which grows logarithmically with n and m which uses a logarithmic encoding of natural numbers and recursion to achieve the same phenomena. We omit the details.

6.4 An Optimised Abstract Machine

To obtain an abstract machine that correctly "models" beta reduction we must eliminate the possibility of pointer chains in the heap. We do so by replacing the (β) rule with an optimised form (β'):

$$\langle\, \Gamma,\ \lambda x.M,\ [\cdot]\, N : S \,\rangle \leadsto_{name} \begin{cases} \langle\, \Gamma,\ M[{}^{N}\!/_{x}],\ S \,\rangle & \text{if } N \text{ is a heap variable} \\ \langle\, \Gamma\{p{=}N\},\ M[{}^{p}\!/_{x}],\ S \,\rangle & \text{otherwise} \end{cases}$$

The key is that the machine now maintains the invariant that for each binding $p{=}N$ in the heap, N is not a heap variable. This ensures that consecutive lookup steps are never performed, and it leads to the following result.

Lemma 11. *If $M{\Downarrow}^n$ then there exist Γ and V such that*

$$\langle\, \emptyset,\ M,\ \epsilon \,\rangle \leadsto_{name}^{\leq 6n} \langle\, \Gamma,\ V,\ \epsilon \,\rangle$$

The proof of Lemma 11 is very similar to the call-by-value case. First we show that the number of reduction steps in the small-step semantics is the same as the number of abstract machine transitions which use one of the rules (β) or (δ).

Lemma 12. *If $M{\Downarrow}^n$ then there exist Γ and V such that*

$$\langle\, \emptyset,\ M,\ \epsilon \,\rangle \leadsto_{name}^{*} \langle\, \Gamma,\ V,\ \epsilon \,\rangle$$

using exactly n applications of (β) and (δ)

The proof of this lemma is by standard techniques relating small-step semantics and abstract machines and is omitted. Next we claim that the total number of abstract machine transitions is at most six times the number of (β) and (δ) transitions.

Lemma 13. *If $\langle\, \emptyset,\ M,\ \epsilon \,\rangle \leadsto_{name}^{m} \langle\, \Gamma,\ V,\ \epsilon \,\rangle$ using n applications of (β) and (δ) then $m \leq 6n$.*

Taken together with Lemma 12 this immediately implies Lemma 11.

Just as in the call-by-value case we generalise Lemma 13 to configurations with a non empty heap and stack.

Lemma 14. *If $\langle\, \Gamma,\ M,\ S \,\rangle \leadsto^{m} \langle\, \Gamma',\ V,\ \epsilon \,\rangle$ using n applications of (β) and (δ) then $m \leq 6n - \lceil M \rceil - \lceil S \rceil + 1$.*

Proof. The measure $\lceil\cdot\rceil$ is the same as that used in the call-by-value case and the proof is very similar. We only consider the case for the lookup rule where we use the invariant property of the heap. Then $\langle\, \Gamma,\ M,\ S\,\rangle \equiv \langle\, \Gamma'\{p = N\},\ p,\ S\,\rangle$ for some p and N. From the induction hypothesis we know that

$$6n - \lceil N\rceil - \lceil S\rceil + 1 \geq m - 1.$$

The required result then follows by the following calculation where the last step uses the fact that N is not a heap variable and thus $\lceil N\rceil = 1$

$$\begin{aligned} 6n - \lceil M\rceil - \lceil S\rceil + 1 &= 6n - \lceil p\rceil - \lceil S\rceil + 1 \\ &= 6n - \lceil S\rceil + 1 \\ &= 6n - \lceil N\rceil - \lceil [\cdot]S\rceil + 1 + \lceil N\rceil \\ &\geq m - 1 + \lceil N\rceil \\ &= m \end{aligned}$$

□

7 Conclusions

We have shown that the core theory of call-by-value functional languages cannot speed up (or slow down) programs by more than a constant factor, and we have stated that the result also holds for call-by-need calculi. In conclusion we reflect on the robustness of this result.

7.1 Machine Variations

The results are clearly dependent on particular implementations. Implementation optimisations which themselves can yield nonlinear speedups can turn innocent transformations into nonlinear speedups or slowdowns. This is illustreded in the call-by-name case. In order to build a machine which matches the beta reduction we were forced to include an optimisation. But the syntactic nature of the optimisation made it very fragile: simply by adding a "tick" to a subterm we can turn off the optimisation and thus get asymptotic slowdown. If we had taken the more naïve implementation model for call-by-name as the basis[3] then the call-by-value result also holds for this variant of call-by-name.

[3] A corresponding high level semantics for the naïve implementation can be obtained by using a modification of beta reduction:

$$(\lambda x.M)\, N \mapsto_{name'} M[{}^{\checkmark}N/_x]$$

7.2 Language Extensions

An obvious question is whether the result is robust under language extension. For this to be a sensible question one must first ask whether the theory itself is *sound* in richer languages. Fortunately Felleisen *et al* [FH92] have shown that the λ_v calculus is sound for quite a number of language extensions including state and control. Given the rather direct operation nature of our proofs (they do not rely on global properties of the language, such as static typing) we have no reason to believe that the same result does not hold in richer languages.

But of course language extensions bring new basic reductions. We believe that the result will also be easy to extend to include basic reductions corresponding to state manipulations – but we have not proved this. It is also straightforward to extend the theory with specific (and useful laws). For example, the following context-distribution law is a strong cost equivalence:

$$\mathbb{E}[\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N] \cong \mathsf{if}\ L\ \mathsf{then}\ \mathbb{E}[M]\ \mathsf{else}\ \mathbb{E}[N]$$

and thus the result is sound if we extend the theory with this rule[4] – or any other rule which is a weak cost equivalence (it need not be a strong cost equivalence as in this example).

We stated the main result for call-by-need calculi. What about other reasonable calculi? The core of the proof rests on the simple cost equivalence ${}^{\checkmark}M \underset{\approx}{\diamond} M$. Although we at first thought that this would hold for *any* reasonable programming calculus, the counterexample for call-by-name shows otherwise. As further work one might consider whether it holds for e.g., object calculi [AC96].

7.3 Theory Extensions: A Program Transformation Perspective

Where do speedups come from? It is folklore that many program transformation techniques cannot produce superlinear speedups. A good example is partial evaluation (except perhaps in rather pathological cases, such as discarding computations by using full beta-reduction), which, viewed abstractly as a source to source transformation, employs little more than basic reduction rules.

One "simple" source of non constant-factor speedups is the avoidance of repeated computation via common subexpression elimination. For example, the transformation rule:

$$\mathsf{let}\ x = M\ \mathsf{in}\ \mathbb{C}[M] \to \mathsf{let}\ x = M\ \mathsf{in}\ \mathbb{C}[x]$$

(where $\mathbb{C}$ does is assumed not to capture free variables in M) is sound for call-by-value, and can achieve non-constant factor speedups. As an example, consider its application to a recursive definition:

$$\begin{aligned} f\ x = {} & \mathsf{if}\ x = 0\ \mathsf{then}\ 1 \\ & \quad \mathsf{else}\ \mathsf{let}\ z = f\ (x-1)\ \mathsf{in}\ z + f\ (x-1) \end{aligned}$$

[4] In the language we have defined, the conditional must be encoded using a strict primitive function, but a suitable version of this rule is easily encoded.

Note that in order to achieve non-constant-factor speedups the subexpression must be a non value (otherwise any instance of the rule is provable in λ_v). It is for this reason that the memoisation in classical partial evaluation or deforestation does *not* achieve more than a constant factor speedup – because the memoisation there typically shares *functions* rather than arbitrary terms.

Further work along these lines would be to extend the constant factor result to a richer class of transformation systems which include forms of memoisation. This would enable us to, or example, answer the question posed in [Jon90] concerning partial evaluation and superlinear speedups.

Acknowledgements

Thanks to Koen Claessen and John Hughes for spotting problems with the naïve implementation model for call-by-name. Thanks also to Dave Schmidt for providing valuable feedback on an earlier draft.

References

[AB97] Z. M. Ariola and S. Blom, *Cyclic lambda calculi*, Proc. TACS'97, LNCS, vol. 1281, Springer-Verlag, February 1997, pp. 77–106.

[AC96] M. Abadi and L. Cardelli, *A theory of objects*, Springer-Verlag, New York, 1996.

[AFM$^+$95] Z. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler, *A call-by-need lambda calculus*, Proc. POPL'95, the 22^{nd} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, January 1995, pp. 233–246.

[AG92] Lars Andersen and Carsten Gomard, *Speedup analysis in partial evaluation: Preliminary results*, Proceedings of the 1992 ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation (San Francisco, U.S.A.), Association for Computing Machinery, June 1992, pp. 1–7.

[BK83] Gerard Boudol and Laurent Kott, *Recursion induction principle revisited*, Theoretical Computer Science **22** (1983), 135–173.

[FH92] Matthias Felleisen and Robert Hieb, *A revised report on the syntactic theories of sequential control and state*, Theoretical Computer Science **103** (1992), no. 2, 235–271.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft, *Partial evaluation and automatic program generation*, Prentice Hall International, International Series in Computer Science, June 1993, ISBN number 0-13-020249-5 (pbk).

[Jon90] Neil D. Jones, *Partial evaluation, self-application and types*, Automata, Languages and Programming, LNCS, vol. 443, Springer-Verlag, 1990, pp. 639–659.

[Jon93] N. D. Jones, *Constant time factors* do *matter*, STOC '93. Symposium on Theory of Computing (Steven Homer, ed.), ACM Press, 1993, pp. 602–611.

[Jon96] Neil D. Jones, *What not to do when writing an interpreter for specialisation*, Partial Evaluation (Olivier Danvy, Robert Glück, and Peter Thiemann, eds.), Lecture Notes in Computer Science, vol. 1110, Springer-Verlag, 1996, pp. 216–237.

[Lan66] P. J. Landin, *The next 700 programming languages*, Communications of the ACM **9** (1966), no. 3, 157–164.

[Mil77] R. Milner, *Fully abstract models of the typed λ-calculus*, Theoretical Computer Science **4** (1977), 1–22.

[MS99] Andrew Moran and David Sands, *Improvement in a lazy context: An operational theory for call-by-need*, Proc. POPL'99, ACM Press, January 1999, pp. 43–56.

[Plo76] G. Plotkin, *Call-by-name, call-by-value and the λ-calculus*, Theoretical Computer Science **1** (1976), no. 1, 125–159.

[Ros98] Eva Rose, *Linear-time hierarchies for a functional language machine model*, Science of Computer Programming **32** (1998), no. 1–3, 109–143, 6th European Symposium on Programming (Linköping, 1996).

[San96] D. Sands, *Proving the correctness of recursion-based automatic program transformations*, Theoretical Computer Science **167** (1996), no. A.

[Zhu94] Hong Zhu, *How powerful are folding/unfolding transformations?*, Journal of Functional Programming **4** (1994), no. 1, 89–112.