
Higher-Order and Symbolic Computation, 13, 217–237, 2000

Linear-Time Self-Interpretation of the Pure Lambda Calculus

TORBEN Æ. MOGENSEN torbenm@diku.dk
DIKU, University of Copenhagen, Denmark

Abstract. We show that linear-time self-interpretation of the pure untyped lambda calculus is possible, in the sense that interpretation has a constant overhead compared to direct execution under various execution models. The present paper shows this result for reduction to weak head normal form under call-by-name, call-by-value and call-by-need.

We use a self-interpreter based on previous work on self-interpretation and partial evaluation of the pure untyped lambda calculus.

We use operational semantics to define each reduction strategy. For each of these we show a simulation lemma that states that each inference step in the evaluation of a term by the operational semantics is simulated by a sequence of steps in evaluation of the self-interpreter applied to the term (using the same operational semantics).

By assigning costs to the inference rules in the operational semantics, we can compare the cost of normal evaluation and self-interpretation. Three different cost-measures are used: number of beta-reductions, cost of a substitution-based implementation (similar to graph reduction) and cost of an environment-based implementation.

For call-by-need we use a non-deterministic semantics, which simplifies the proof considerably.

Keywords: programs as data, lambda calculus, self-interpretation, reduction order, cost measures, semantics

1. Program and data representation

In order to talk about self-interpretation of the pure lambda calculus, we must distinguish between a program as a function and as as data (syntax). Since we want to use programs as input to a program written in the lambda calculus, we consider how to represent program syntax as data acceptable by a lambda-calculus program. In lambda calculus this is less than obvious, as there is no clear distinction between programs and data. The obvious choice is to represent a program by itself (as it is already a lambda term). This would make self-interpretation trivial, but it is on closer inspection not a very good choice:

- Using this trivial representation, it is not possible to discriminate different programs by operations (inside the lambda calculus) on their representations. Nor is it possible to take a syntax tree apart into its constituents.
- If values in a data-domain are represented by lambda terms not in normal form, some of the execution time of running a program on this data is used for reductions on this term itself, making it hard to reason about complexity.

Hence, we want representations of data as lambda terms to have two properties:

i. It must be possible inside the lambda calculus to discriminate representations of different values (i.e. programs). In other words, for any given data domain a lambda term must exist that given representations of two values in that domain reduce to $\lambda xy.x$ if the values are identical and $\lambda xy.y$ if they are not. Furthermore, it must be possible to take a composite value apart into its constituents.
ii. The representation of a value (or program) must be a normal-form lambda term.

Note that programs need not be in normal form, but the representations of these when they are used as input to other programs (such as a self-interpreter) must be.

Various representations can be used for this, see, e.g., [7, 8], and [9]. In general there is a choice between "Church-style" and "standard style" representations, named after Church numerals and standard numerals, see [1], Section 6.4.

In Church-style representations, recursion over the structure of the data is encoded in the representation of the data (as for Church-numbers). In standard representation this is not the case, and explicit recursion (e.g., by use of a Y combinator) is needed in the program to traverse the data.

Each style of representation has advantages and disadvantages. In particular, it is difficult to take a Church-style term apart into its components (e.g., take the predecessor of a Church-number), which for some applications offsets the ease of doing traversals. However, it is generally possible to convert between the two styles of representations inside the lambda calculus (see Appendix A), so we will use whatever style suits us best for each given instance.

As stated above, we want to be able to discriminate different values by inspecting their representations (from within the lambda calculus). In particular, we want this to be true for representations of programs. We just have to decide what we mean by "different" programs. In lambda calculus, we typically operate with two types of equivalence: alpha equivalence and beta equivalence.[1] So for "different" we can substitute either "not alpha equivalent" or "not beta equivalent". In the latter case we should, by inspecting the representations of two terms, be able to determine if they are (not) beta equivalent. Since this is not computable, we clearly cannot do this unless the representation function by an oracle chooses a representative for every beta-equivalence class. For this reason (as well as a wish to study syntactic properties) we use alpha equivalence as the basic, through the representations, decidable equivalence. Note that while this means that we want to be able to *determine* alpha equivalence by inspecting representations, we do not require alpha equivalent terms to map to *identical* representations.

In addition to distinguishing between Church-style and standard-style representations of data, we can also choose from first-order and higher-order syntax. Since all data is represented by lambda terms, which are inherently higher-order, this may seem like an odd distinction. However, we use the term higher-order syntax in the same way as in [10] to mean that variables in the represented term are represented by variables in the representing term, rather than by naming or numbering. The main advantage of higher-order syntax is that substitution of a variable by a value can be trivially implemented by a beta-reduction. Furthermore, alpha-equivalence is automatically preserved in the representation,

i.e, alpha-equivalent terms map to alpha-equivalent representations. On the other hand, treatment of free variables can become slightly more complex.

Since we anyway restrict ourselves to closed lambda terms, we can without qualms use the higher-order, Church-style representation defined in [9]:

$$
\begin{array}{ll}
\ulcorner M \urcorner & \equiv \lambda a.\lambda b.\, \overline{M} \\
\quad \textit{where} & \\
\bar{x} & \equiv x \\
\overline{PQ} & \equiv a\,\overline{P}\,\overline{Q} \\
\overline{\lambda x.P} & \equiv b\,\lambda x.\overline{P}
\end{array}
$$

where M has been renamed so the variables a and b do not occur anywhere. $\equiv$ is alpha-equivalence. The combination of Church-style representation and higher-order syntax yields an exceedingly simple self-interpreter:

$$\textit{selfint} \equiv \lambda m.m\ I\ I$$

where $I \equiv \lambda x.x$ is the identity combinator.

The proof that

$$\textit{selfint}\ \ulcorner M \urcorner \twoheadrightarrow M$$

where $\twoheadrightarrow$ means "reduces in zero or more beta reductions to" is trivial: Reduction of the topmost application, the application of $\ulcorner M \urcorner$ to two I's and reduction of the application of these I's to their arguments once they have been substituted for a and b inside $\ulcorner M \urcorner$ will yield M.

Note, however, that the reductions above does not correspond to any commonly used reduction order. A consequence is that reduction of $\textit{selfint}\ulcorner M \urcorner$ to weak head-normal form (WHNF) using a specific reduction order does not necessarily yield the same WHNF as is obtained by reducing M by the same reduction order (reduction to full normal form will, however, obviously yield identical terms).

The issue becomes further muddled when using operations semantics (as we will do below) where WHNF's are represented by closures. We will for each reduction order state a relation between closures obtained through normal evaluation and those obtained by self-interpretation and compare the costs of obtaining these. Apart from that, we will not address this issue further.

The self-interpreter is so simple that it feels like cheating. The reasons for the simplicity are the use of a Church-style representation which encode recursion over syntax in the representation and the use of higher-order abstract syntax which uses bindings to represent bindings. The representation does, however, fulfil the criteria we set up: $\ulcorner M \urcorner$ is clearly in normal form and we can in the lambda calculus encode functions for testing alpha-equivalence and for taking apart and building representations of lambda terms. See Appendix A for details.

2. Linearity of self-interpretation

Evaluation time in the lambda-calculus depends on the reduction order, so we need to specify which order is used when we make any claims about relative speeds. We will do so below for reduction to WHNF using call-by-name, call-by-value and call by need. To make each of these reduction orders explicit, we define them by operational semantics in the style of Plotkin [11], as described, e.g., in [4].

Additionally, we define various measures of cost of evaluation using these reduction strategies. This is done by assigning cost to the inference rules in the semantics, such that the total cost of evaluation is obtained by adding the costs of the rules used to build the evaluation tree.

Note that our claim is not that we can do self-interpretation in time linear to the size of the interpreted program, but that this is linear in the time used for normal evaluation. Hence, what we measure is the interpretation overhead, and our claim is that this is bounded by a small constant that depends on the choice of reduction order and cost measure.

3. Linear-time self-interpretation using call-by-name reduction

Call-by-name evaluation can be described by the inference rules in figure 1.

We can define various cost measures by assigning costs to uses of the inference rules in an evaluation tree. For example, we can count beta reductions by letting each use of the (*BETA*) rule count 1 and not charge anything for the other rules. But we can also define more fine-grained (and more realistic) cost measures by assigning different costs.

We start by showing (in figure 2) how the inference rules can be used to derive the initial stages of self-interpretation of a term M.

We will use the evaluation tree from figure 2 later in the proof, and we will define $\bar{\rho}$ by the rules:

$$\frac{}{\overline{[]} = \rho_3}$$

$$\overline{\rho[x \mapsto (S, \rho')] = \bar{\rho}[x \mapsto (\overline{S}, \overline{\rho'})]}$$

$$\rho \vdash \lambda x.M \Rightarrow (\lambda x.M,\ \rho) \quad \textit{(LAMBDA)}$$

$$\frac{\rho' \vdash M \Rightarrow W}{\rho \vdash x \Rightarrow W} \quad \text{where } \rho(x) = (M, \rho') \quad \textit{(VAR)}$$

$$\frac{\rho \vdash M \Rightarrow (\lambda x.M',\ \rho') \qquad \rho'[x \mapsto (N, \rho)] \vdash M' \Rightarrow W}{\rho \vdash M\ N \Rightarrow W} \quad \textit{(BETA)}$$

Figure 1. Call-by-name evaluation.

$$\frac{\rho_0 \vdash \mathit{selfint} \Rightarrow (\mathit{selfint}, \rho_0) \qquad (*)}{\rho_0 \vdash \mathit{selfint}\ \ulcorner M \urcorner \Rightarrow W}$$

where $(*)$ is the inference tree

$$\dfrac{\dfrac{\dfrac{\rho_0 \vdash \ulcorner M \urcorner \Rightarrow (\ulcorner M \urcorner, \rho_0)}{\rho_1 \vdash m \Rightarrow (\ulcorner M \urcorner, \rho_0)} \qquad \rho_2 \vdash \lambda b.\overline{M} \Rightarrow (\lambda b.\overline{M}, \rho_2)}{\rho_1 \vdash m\ I \Rightarrow (\lambda b.\overline{M}, \rho_2)} \qquad \dfrac{\cdots}{\rho_3 \vdash \overline{M} \Rightarrow W}}{\rho_1 \vdash m\ I\ I \Rightarrow W}$$

and

$$\begin{aligned}\rho_0 &= [] \\ \rho_1 &= [m \mapsto (\ulcorner M \urcorner, \rho_0)] \\ \rho_2 &= [a \mapsto (I, \rho_1)] \\ \rho_3 &= [a \mapsto (I, \rho_1),\ b \mapsto (I, \rho_1)]\end{aligned}$$

Figure 2. Initial stages of call-by-name self-interpretation.

where ρ_3 is as defined in figure 2, where the M referred to in ρ_1 is be the entire term being interpreted. The actual value for this binding is, however, irrelevant for the later proofs as it is only used in the initial stages shown in figure 2. Note that $|\bar{\rho}| = |\rho| + 2$, where $|\bar{\rho}|$ is the number of variables bound in ρ.

We need a *simulation lemma*:

Lemma 1. *If from the call-by-name inference rules we can derive the evaluation* $\rho \vdash N \Rightarrow (\lambda y.W,\ \rho')$ *then we can also derive the evaluation* $\bar{\rho} \vdash \overline{N} \Rightarrow (\lambda y.\overline{W},\ \overline{\rho'})$.

which we prove in figure 3. We use in this (and the following proofs) a notation where "$\cdots$" refers to an unspecified proof tree. This is to indicate where an induction step is used: If normal evaluation has a proof tree indicated by "$\cdots$", we replace this in the simulation by a proof tree that by induction is assumed to exist. This proof tree is in the simulation also indicated by "$\cdots$". Semantic variables (place holders) in the conclusion of a rule where the premise is "$\cdots$", can be considered existentially quantified, like variables in the premise of an inference rule typically are.

By assigning costs to the inference rules, we can count the costs for normal evaluation and self-interpretation and hence prove linear-time self-interpretation. We start by counting beta reductions. For this, we let each use of the (*BETA*) rule count 1 and the other rules count 0.

The tree for the initial stages of self-interpretation uses the beta rule three times, so this tree has cost 3. For the remainder of the computations we use this lemma:

Proof. We prove lemma 1 by induction over the structure of the evaluation tree with N at its root:
$N = x$: Let $\rho(x) = (S,\ \rho'')$. For normal evaluation we have

$$\dfrac{\dfrac{\cdots}{\rho'' \vdash S \Rightarrow (\lambda y.W,\ \rho')}}{\rho \vdash x \Rightarrow (\lambda y.W,\ \rho')}$$

and for self-interpretation, the corresponding tree is

$$\dfrac{\dfrac{\cdots}{\overline{\rho''} \vdash \overline{S} \Rightarrow (\lambda y.\overline{W},\ \overline{\rho'})}}{\overline{\rho} \vdash x \Rightarrow (\lambda y.\overline{W},\ \overline{\rho'})}$$

$N = \lambda y.W$: $\rho \vdash N \Rightarrow (\lambda y.W,\ \rho)$ is a leaf tree. $\overline{N} = b\ (\lambda y.\overline{W})$, so for self-interpretation we get the tree

$$\dfrac{\dfrac{\rho_1 \vdash I \Rightarrow (I, \rho_1)}{\overline{\rho} \vdash b \Rightarrow (\lambda z.z, \rho_1)} \quad \dfrac{\overline{\rho} \vdash \lambda y.\overline{W} \Rightarrow (\lambda y.\overline{W}, \overline{\rho})}{\rho_1[z \mapsto (\lambda y.\overline{W}, \overline{\rho})] \vdash z \Rightarrow (\lambda y.\overline{W}, \overline{\rho})}}{\overline{\rho} \vdash b\ (\lambda y.\overline{W}) \Rightarrow (\lambda y.\overline{W}, \overline{\rho})}$$

$N = N_1\ N_2$: The normal evaluation tree is

$$\dfrac{\dfrac{\cdots}{\rho \vdash N_1 \Rightarrow (\lambda v.N_3,\ \rho'')} \quad \dfrac{\cdots}{\rho''[v \mapsto (N_2, \rho)] \vdash N_3 \Rightarrow (\lambda y.W,\ \rho')}}{\rho \vdash N_1\ N_2 \Rightarrow (\lambda y.W,\ \rho')}$$

We have $\overline{N} = a\ \overline{N_1}\ \overline{N_2}$, so we get (by induction) the following tree for self-interpretation

$$\dfrac{\dfrac{\dfrac{\rho_1 \vdash I \Rightarrow (I, \rho_1)}{\overline{\rho} \vdash a \Rightarrow (\lambda z.z, \rho_1)} \quad \dfrac{\dfrac{\cdots}{\overline{\rho} \vdash \overline{N_1} \Rightarrow (\lambda v.\overline{N_3},\ \overline{\rho''})}}{\rho_1[z \mapsto (\overline{N_1},\ \overline{\rho})] \vdash z \Rightarrow (\lambda v.\overline{N_3},\ \overline{\rho''})}}{\overline{\rho} \vdash a\ \overline{N_1} \Rightarrow (\lambda v.\overline{N_3},\ \overline{\rho''})}\ (*)}{\overline{\rho} \vdash a\ \overline{N_1}\ \overline{N_2} \Rightarrow (\lambda y.\overline{W},\ \overline{\rho'})}$$

where $(*)$ is the inference tree

$$\dfrac{\dfrac{\cdots}{\overline{\rho''}[v \mapsto (\overline{N_2},\ \overline{\rho})] \vdash \overline{N_3} \Rightarrow (\lambda y.\overline{W},\ \overline{\rho'})}}{}$$

□

Figure 3. Proof of Lemma 1.

Lemma 2. *If derivation of* $\rho \vdash N \Rightarrow (\lambda y.W,\ \rho')$ *uses n beta-reductions, then derivation of* $\bar{\rho} \vdash \overline{N} \Rightarrow (\lambda y.\overline{W},\ \overline{\rho'})$ *uses* $3n + 1$ *beta-reductions.*

Proof: The proof is done by induction over the structure of the evaluation tree, using the proof of Lemma 1. as skeleton.

$N = x$: Neither normal evaluation nor self-interpretation uses the (*BETA*) rule, so the result follows by induction on the subtree.

$N = \lambda y \cdot W$: The normal evaluation tree has cost 0 while self-interpretation uses the (*BETA*) rule once. Since $3 \cdot 0 + 1 = 1$, we are done.

$N = N_1\, N_2$: Assuming the subtrees have costs k_1 and k_2 respectively, the total cost of normal evaluation is $k_1 + k_2 + 1$. By induction, the cost for the subtrees for self-interpretation are $3k_1 + 1$ and $3k_2 + 1$ and the tree uses (*BETA*) twice, so the total cost is $3k_1 + 3k_2 + 4 = 3(k_1 + k_2 + 1) + 1$, which is what we want. □

By adding the cost for the initial states of self-interpretation, we are ready for the theorem:

Theorem 3. *If a closed term M via the call-by-name semantics evaluates to a weak head normal form* (*WHNF*) *using n beta reductions, then selfint* $\lceil M \rceil$ *evaluates to a WHNF using* $3n + 4$ *beta reductions.*

3.1. A more realistic cost measure

Just counting beta reductions is a fairly crude way of measuring the cost of reduction of lambda terms. In this section and the next we will study measures that emulate common methods for implementing functional languages.

The first of these is simplified graph rewriting (actually, term rewriting as we don't exploit sharing or circularity). In graph rewriting, a beta-reduction is implemented by making a new copy of the body of the function and inserting the argument in place of the variables.[2] This has a cost which is proportional to the size of the function that is applied. Hence, we will use a cost measure that for each use of the (*BETA*) rule has a cost equal to the size of the function $(\lambda x.M')$ that is applied. The other rules still count 0, as the use of environments and closures in the inference rules do not directly correspond to actions in graph rewriting. Instead, we will treat each closure (P, ρ) as the term obtained by substituting the free variables in P by the values bound to them in ρ, after the same has been done recursively to these values. More formally, we define the function *unfold* by:

$$\begin{aligned} &\mathit{unfold}(P, []) &&= P \\ &\mathit{unfold}(P, \rho[x \mapsto (Q, \rho')]) &&= \mathit{unfold}(P, \rho)[x \backslash \mathit{unfold}(Q, \rho')] \end{aligned}$$

We need a small lemma

Lemma 4. $\mathit{unfold}(\overline{P}, \bar{\rho}) = \overline{\mathit{unfold}(P, \rho)}[a \backslash I][b \backslash I]$.

Proof: We prove this by induction over the definition of *unfold*:

$unfold(P, []) = P$:

$$
\begin{aligned}
&unfold(\overline{P}, \overline{[]}) \\
&= unfold(\overline{P}, \rho_3) \\
&= unfold(\overline{P}, \rho_2)[b\backslash unfold(I, \rho_1)] \\
&= unfold(\overline{P}, \rho_2)[b\backslash I] \\
&= \underline{unfold}(\overline{P}[a\backslash unfold(I, \rho_1)], [])[b\backslash I] \\
&= \overline{P}[a\backslash I][b\backslash I]
\end{aligned}
$$

$unfold(P, \rho[x \mapsto (Q, \rho')]) = unfold(P, \rho)[x\backslash unfold(Q, \rho'')]$:

$$
\begin{aligned}
&unfold(\overline{P}, \overline{\rho[x \mapsto (Q, \rho')]}) \\
&= unfold(\overline{P}, \bar{\rho}[x \mapsto (\bar{Q}, \overline{\rho'})]) \\
&\qquad\qquad \text{by definition of } \bar{\rho} \\
&= unfold(\overline{P}, \bar{\rho})[x\backslash unfold(\bar{Q}, \overline{\rho'})] \\
&\qquad\qquad \text{by definition of } unfold \\
&= \overline{unfold(P, \rho)}[a\backslash I][b\backslash I][x\backslash \overline{unfold(Q, \rho')}[a\backslash I][b\backslash I]] \\
&\qquad\qquad \text{by induction} \\
&= \overline{unfold(P, \rho)}[x\backslash \overline{unfold(Q, \rho')}][a\backslash I][b\backslash I] \\
&= \overline{unfold(P, \rho)[x\backslash unfold(Q, \rho')]}[a\backslash I][b\backslash I]
\end{aligned}
$$

□

We count the size of a term as the number of nodes in the syntax tree, i.e. one for each variable occurrence plus one for each application and one for each abstraction. It is easy to see that the size of $\overline{P}[a\backslash I][b\backslash I]$ is strictly less than 4 times the size of P.

We first count the cost of the initial part of the tree to be $|selfint| = 8$ for the first beta reduction, $|\lceil M \rceil| < 3|M|$ for the second and the size of $\lambda b.\, \overline{M}$ with a replaced by I ($< 4|M|$) for the third, for a total cost less than $7|M| + 8$.

We now proceed with the lemma

Lemma 5. *If derivation of* $\rho \vdash N \Rightarrow (\lambda y.W,\ \rho')$ *has cost* c (*using the substitution-based cost measure*), *then derivation of* $\bar{\rho} \vdash \overline{N} \Rightarrow (\lambda y.\overline{W},\ \overline{\rho'})$ *has cost at most* $4c + 2$.

Proof: Again we prove this by induction following the structure of the proof for Lemma 1.

$N = x$: Neither normal evaluation nor self-interpretation uses the (*BETA*) rule, so the result follows by induction on the subtrees.

$N = \lambda y.W$: The normal evaluation tree has cost 0 while self-interpretation uses the (*BETA*) rule once. The applied function is $\lambda z.z$ which has size 2, so we have what we need.

$N = N_1\, N_2$: Assuming the subtrees have costs k_1 and k_2 respectively, the total cost of normal evaluation is $k_1 + k_2 + s$, where s is the size of $unfold(\lambda v.N_3,\ \rho'')$. By induction, the cost for the subtrees for self-interpretation are at most $4k_1 + 2$ and $4k_2 + 2$. The tree uses (*BETA*) twice, once for the function $\lambda z.z$ (size 2) and once for $unfold\ (\lambda v.\bar{N}_3,\ \bar{\rho}'') = \lambda v.\overline{unfold(N_3,\ \rho'')}[a\backslash I][b\backslash I]$.

Since the size of $\overline{unfold(N_3,\ \rho'')}[a\backslash I][b\backslash I]$ is strictly less than 4 times the size of $unfold(N_3,\ \rho'')$, we have that the size of $\lambda v.\overline{unfold(N_3,\ \rho'')}[a\backslash I][b\backslash I]$ is at most

$4|\mathit{unfold}(N_3,\ \rho'')| - 1 + 1 = 4(s-1)$. Hence, we have a total cost bounded by $4k_1 + 2 + 4k_2 + 2 + 2 + 4(s-1) \leq 4(k_1 + k_2 + s) + 2$, which is what we needed. □

By combining Lemma 5 with the start-up cost of $7|M| + 8$, we get the theorem

Theorem 6. *If a closed term M via the call-by-name semantics evaluates to a WHNF in cost c (using the substitution-based cost measure), selfint $\ulcorner M \urcorner$ evaluates to a WHNF in cost at most $4c + 7|M| + 10$.*

The start-up cost proportional to the size of M is unavoidable, regardless of how lambda terms are represented and how the self-interpreter works. We required representations to be in normal form, so to perform any evaluation that depends on the representation, we will have to apply the representation to one or more arguments, which by our measure has a cost proportional to the size of the representation, which cannot be less than linear in the size of the term.

3.2. Environment-based cost

Another common method for implementing call-by-name lambda calculus is using environments and closures, much as indicated by the inference rules. The cost measure used for an environment-based implementation depends on how the environments are implemented. Typical data structures are linked lists and frames. Using a linked list, a new variable is added to the front of the list at unit cost, but accessing a variable has a cost that depends on the position of the variable in the environment. If frames are used, a new extended copy of the environment is built every time a new variable is added to it. This has cost proportional to the size of the new environment, but accessing a variable in the environment is now unit cost.

With the chosen interpreter, we can not get linear-time self-interpretation if linked-list environments are used, as looking up a or b has a cost that depends on the size of the environment, which ultimately depends on the size of the program. We shall, however, see that we do get linear-time self-interpretation with frames.

Our cost measure now counts each use of the (*VAR*) or (*LAMBDA*) rule as 1 and each use of the (*BETA*) rule as the size of the new frame, i.e. $|\rho'| + 1$.

We first note that the cost of the initial part of the evaluation tree is 8. We then state and prove the following lemma:

Lemma 7. *If derivation of $\rho \vdash N \Rightarrow (\lambda y.W,\ \rho')$ has cost c (using the environment-based cost measure), then derivation of $\bar{\rho} \vdash \overline{N} \Rightarrow (\lambda y.\overline{W},\ \overline{\rho'})$ has cost at most $8c$.*

Proof: $N = x$: Both normal evaluation and self-interpretation use the (*VAR*) rule once, so if the cost of evaluating the contents of the variable is k, the total evaluation cost is $k+1$. By induction, self-interpretation of the contents costs at most $8k$, for a total self-interpretation cost of $8k + 1$, which is less than the $8(k + 1)$ limit.

$N = \lambda y.W$: The normal evaluation tree has cost 1, for a single use of the (*VAR*) rule. Self-interpretation uses (*VAR*) and (*LAMBDA*) twice each and the (*BETA*) rule once. The

size of the expanded environment is 2, so we have a total cost of 6, which is less than 8 times the cost of normal evaluation.

$N = N_1\,N_2$: Assuming the subtrees have costs k_1 and k_2 respectively, the total cost of normal evaluation is $k_1 + k_2 + |\rho''| + 1$. By induction, the cost for the subtrees for self-interpretation are at most $8k_1$ and $8k_2$. The tree uses (*VAR*) twice, (*LAMBDA*) once and (*BETA*) twice, once for the function $\lambda z.z$ (where the size of the expanded environment is 2) and once for $(\lambda v.\overline{N_3},\ \overline{\rho''})$. Since $|\overline{\rho''}| = |\rho''| + 2$, the total cost is bounded by $8k_1 + 8k_2 + 2 + 1 + 2 + |\rho''| + 3 = 8k_1 + 8k_2 + |\rho''| + 8$, which is less than the budget of $8(k_1 + k_2 + |\rho''| + 1)$. □

By adding the start-up cost of 8 to the cost found in Lemma 7. we get:

Theorem 8. *If a closed term M evaluates to a WHNF in cost c (using the environment-based cost measure), selfint $\lceil M \rceil$ evaluates to a WHNF in cost at most $8c + 8$.*

4. Linear-time self-interpretation using call-by-value reduction

We define call-by-value reduction by the inference rules in figure 4.

We again start by showing (in figure 5) how the inference rules can be used to derive the initial stages of self-interpretation of a term M. Note that the environments are the same as in the call-by-name case. We will, however, slightly change definition of $\bar{\rho}$ to reflect that variables are bound to values in WHNF:

$$\begin{aligned}\overline{[]} &= \rho_3\\ \overline{\rho[x \mapsto (\lambda x.P, \rho')]} &= \bar{\rho}[x \mapsto (\lambda x.\overline{P}, \overline{\rho'})]\end{aligned}$$

Again, the M referred to in the environments is the entire term being interpreted and, again, this is only relevant for the initial stages shown in figure 5.

We first define a simulation lemma for call-by-value:

$$\rho \vdash \lambda x.M \Rightarrow (\lambda x.M,\ \rho) \quad \textit{(LAMBDA)}$$

$$\rho \vdash x \Rightarrow \rho(x) \quad \textit{(VARV)}$$

$$\frac{\rho \vdash M \Rightarrow (\lambda x.M',\ \rho') \quad \rho \vdash N \Rightarrow V \quad \rho'[x \mapsto V] \vdash M' \Rightarrow W}{\rho \vdash M\ N \Rightarrow W} \quad \textit{(BETAV)}$$

Figure 4. Inference rules for call-by-value evaluation.

$$\dfrac{\rho_0 \vdash \mathit{selfint} \Rightarrow (\mathit{selfint}, \rho_0) \qquad \dfrac{(*) \quad \rho_1 \vdash I \Rightarrow (I, \rho_1) \qquad \dfrac{\ldots}{\rho_3 \vdash \overline{M} \Rightarrow W}}{\rho_1 \vdash m\ I\ I \Rightarrow W}}{\rho_0 \vdash \mathit{selfint}\ \ulcorner M \urcorner \Rightarrow W}$$

where $(*)$ is the tree

$$\dfrac{\rho_1 \vdash m \Rightarrow (\ulcorner M \urcorner, \rho_0) \quad \rho_1 \vdash I \Rightarrow (I, \rho_1) \quad \rho_2 \vdash \lambda b.\overline{M} \Rightarrow (\lambda b.\overline{M}, \rho_2)}{\rho_1 \vdash m\ I \Rightarrow (\lambda b.\overline{M}, \rho_2)}$$

and

$$\begin{aligned}\rho_0 &= [] \\ \rho_1 &= [m \mapsto (\ulcorner M \urcorner, \rho_0)] \\ \rho_2 &= [a \mapsto (I, \rho_1)] \\ \rho_3 &= [a \mapsto (I, \rho_1),\ b \mapsto (I, \rho_1]\end{aligned}$$

Figure 5. Initial stages of call-by-value self-interpretation.

Lemma 9. *If we, using the call-by-value inference rules, can derive* $\rho \vdash N \Rightarrow (\lambda y.W,\ \rho')$ *then we can also derive* $\bar{\rho} \vdash \overline{N} \Rightarrow (\lambda y.\overline{W},\ \overline{\rho'})$.

which we prove in figure 6.

Again, we assign different costs to the rules to obtain linear-time self-interpretation results. We start by counting beta-reductions.

The initial part of the tree uses 3 beta reductions. For the remainder we use a lemma like the one for call-by-name:

Lemma 10. *If call-by-value derivation of* $\rho \vdash N \Rightarrow (\lambda y.W,\ \rho')$ *uses n beta-reductions, then call-by-value derivation of* $\bar{\rho} \vdash \overline{N} \Rightarrow (\lambda y.\overline{W},\ \overline{\rho'})$ *uses at most* $4n + 1$ *beta-reductions.*

Proof: We will use the structure of Lemma 9 for proving this.

$N = x$: Neither normal evaluation nor self-interpretation use beta reductions.

$N = \lambda y.W$: The normal evaluation tree uses 0 reductions while self-interpretation uses the (*BETA*) rule once, giving $4 \cdot 0 + 1$, as we needed.

$N = N_1\ N_2$: Assuming the subtrees use k_1, k_2 and k_3 beta reductions respectively, the total number of reductions in normal evaluation is $k_1 + k_2 + k_3 + 1$. By induction, the subtrees for self-interpretation use at most $4k_1 + 1, 4k_2 + 1$ and $4k_3 + 1$ reductions. The tree uses (*BETA*) twice, so the total reduction count is bounded by $4k_1 + 4k_2 + 4k_3 + 5 = 4(k_1 + k_2 + k_3 + 1) + 1$, which is what we want. □

By adding the cost for the initial states of self-interpretation, we are ready for the theorem:

Proof. We prove lemma 9 by induction over the structure of the evaluation tree with N at its root:

$N = x$: For normal evaluation we have

$$\rho \vdash x \Rightarrow (\lambda y.W,\ \rho')$$

For self-interpretation, the corresponding tree is (by definition of $\overline{\rho}$)

$$\overline{\rho} \vdash x \Rightarrow (\lambda y.\overline{W},\ \overline{\rho'})$$

$N = \lambda y.W$: $\rho \vdash N \Rightarrow (\lambda y.W,\ \rho)$ is a leaf tree. $\overline{N} = b\ (\lambda y.\overline{W})$, so for self-interpretation we get the tree

$$\frac{\overline{\rho} \vdash b \Rightarrow (\lambda z.z, \rho_1) \quad \rho_1[z \mapsto (\lambda y.\overline{W}, \overline{\rho})] \vdash z \Rightarrow (\lambda y.\overline{W}, \overline{\rho})}{\overline{\rho} \vdash b\ (\lambda y.\overline{W}) \Rightarrow (\lambda y.\overline{W}, \overline{\rho})}$$

$N = N_1\ N_2$: The normal evaluation tree is

$$\frac{\dfrac{\cdots}{\rho \vdash N_1 \Rightarrow (\lambda v.N_3, \rho'')} \quad \dfrac{\cdots}{\rho \vdash N_2 \Rightarrow (\lambda w.N_4, \rho''')} \quad (*)}{\rho \vdash N_1\ N_2 \Rightarrow (\lambda y.W, \rho')}$$

where $(*)$ is the inference tree

$$\frac{\cdots}{\rho''[v \mapsto (\lambda w.N_4, \rho''')] \vdash N_3 \Rightarrow (\lambda y.W, \rho')}$$

We have $\overline{N} = a\ \overline{N_1}\ \overline{N_2}$, so we get (by induction) the following tree for self-interpretation

$$\frac{(*)\ \dfrac{\cdots}{\overline{\rho} \vdash \overline{N_2} \Rightarrow (\lambda w.\overline{N_4},\ \overline{\rho'''})} \quad \dfrac{\cdots}{\overline{\rho''}[v \mapsto (\lambda w.\overline{N_4},\ \overline{\rho'''})] \vdash \overline{N_3} \Rightarrow (\lambda y.\overline{W},\ \overline{\rho'})}}{\overline{\rho} \vdash a\ \overline{N_1}\ \overline{N_2} \Rightarrow (\lambda y.\overline{W},\ \overline{\rho'})}$$

where $(*)$ is the tree

$$\frac{\overline{\rho} \vdash a \Rightarrow (\lambda z.z, \rho_1) \quad \dfrac{\cdots}{\overline{\rho} \vdash \overline{N_1} \Rightarrow (\lambda v.\overline{N_3},\ \overline{\rho''})} \quad (**)}{\overline{\rho} \vdash a\ \overline{N_1} \Rightarrow (\lambda v.\overline{N_3},\ \overline{\rho''})}$$

and $(**)$ is $\quad \rho_1[z \mapsto (\lambda v.\overline{N_3},\ \overline{\rho''})] \vdash z \Rightarrow (\lambda v.\overline{N_3},\ \overline{\rho''})$.

□

Figure 6. Proof of Lemma 9.

Theorem 11. *If a closed term M evaluates to a WHNF using n call-by-value beta reductions, selfint $\ulcorner M \urcorner$ evaluates to a WHNF using at most $4n+4$ call-by-value beta reductions.*

4.1. Substitution-based cost

Again we want to base the cost of a beta reduction on the size of the function, and again we consider a value $(\lambda y.P, \rho)$ to represent the term $\mathit{unfold}(\lambda y.P, \rho)$. We need a variant of Lemma 4, using the new definition of $\bar{\rho}$. We do not get equality, as we did in Lemma 4, as some terms T may be replaced by (IT). We define $P \preceq Q$ to mean that some subterms T in P may be replaced by (IT) in Q and use this in the formulation of the new lemma. Note that size of P is no larger than the size of Q.

Lemma 12. $\mathit{unfold}(\lambda y.\overline{P}, \bar{\rho}) \preceq \lambda y.\overline{\mathit{unfold}(P, \rho)}[a\backslash I][b\backslash I]$.

where $P \preceq Q$ means that some subterms T in P may be replaced by (IT) in Q. Hence, the size of P is no larger than the size of Q.

Proof: We prove Lemma 12 similarly to the way we proved Lemma 4:
$\mathit{unfold}(\lambda y.P, []) = \lambda y.P$:

$$
\begin{aligned}
&\mathit{unfold}(\lambda y.\overline{P}, \overline{[]}) \\
&= \mathit{unfold}(\lambda y.\overline{P}, \rho_3) \\
&= \mathit{unfold}(\lambda y.\overline{P}, \rho_2)[b\backslash \mathit{unfold}(I, \rho_1)] \\
&= \mathit{unfold}(\lambda y.\overline{P}, \rho_2)[b\backslash I] \\
&= \mathit{unfold}(\lambda y.\overline{P}[a\backslash \mathit{unfold}(I, \rho_1)], [])[b\backslash I] \\
&= \lambda y.\overline{P}[a\backslash I][b\backslash I]
\end{aligned}
$$

$\mathit{unfold}(\lambda y.P, \rho[x \mapsto (\lambda z.Q, \rho')]) = \mathit{unfold}(\lambda y.P, \rho)[x\backslash \mathit{unfold}(\lambda z.Q, \rho'')]$:

$$
\begin{aligned}
&\mathit{unfold}(\lambda y.\overline{P}, \overline{\rho[x \mapsto (\lambda z.Q, \rho')]}) \\
&= \mathit{unfold}(\lambda y.\overline{P}, \bar{\rho}[x \mapsto (\lambda z.\bar{Q}, \bar{\rho}')]) \\
&\qquad\qquad \text{by definition of } \bar{\rho} \\
&= \mathit{unfold}(\lambda y.\overline{P}, \bar{\rho})[x\backslash \mathit{unfold}(\lambda z.\bar{Q}, \bar{\rho}')] \\
&\qquad\qquad \text{by definition of } \mathit{unfold} \\
&\preceq \lambda y.\overline{\mathit{unfold}(P, \rho)}[a\backslash I][b\backslash I][x\backslash \lambda z.\overline{\mathit{unfold}(Q, \rho')}[a\backslash I][b\backslash I]] \\
&\qquad\qquad \text{by induction} \\
&= \lambda y.\overline{\mathit{unfold}(P, \rho)}[x\backslash \lambda z.\overline{\mathit{unfold}(Q, \rho')}][a\backslash I][b\backslash I] \\
&\preceq \lambda y.\overline{unfold(P, \rho)[x\backslash \mathit{unfold}(\lambda y.Q, \rho')]}[a\backslash I][b\backslash I]
\end{aligned}
$$

Since the size of $\overline{P}[a\backslash I][b\backslash I]$ is strictly less than 4 times the size of P, we see that $|\mathit{unfold}(\lambda y.\overline{P}, \bar{\rho})| \leq |\lambda y.\overline{\mathit{unfold}(P, \rho)}[a\backslash I][b\backslash I]| < 1 + 4|\mathit{unfold}(P, \rho)|$. □

We count the cost of the initial part of the tree to be at most $7|M| + 8$, just as for the call-by-name case. For the rest, we use the lemma.

Lemma 13. *If call-by-value derivation of* $\rho \vdash N \Rightarrow (\lambda y.W,\ \rho')$ *has cost* c (*using the substitution-based cost measure*), *then call-by-value derivation of* $\bar{\rho} \vdash \overline{N} \Rightarrow (\lambda y.\overline{W},\ \overline{\rho'})$ *has cost at most* $5c + 2$.

Proof: $N = x$: Both normal evaluation and self-interpretation has cost 0.

$N = \lambda y.W$: The normal evaluation tree has cost 0 while self-interpretation uses the (*BETA*) rule once for the term $\lambda z.z$, which has size 2, giving $5 \cdot 0 + 2$, as we needed.

$N = N_1\ N_2$: Assuming the subtrees have costs k_1, k_2 and k_3 respectively, the total cost of normal evaluation is $k_1 + k_2 + k_3 + s$, where s is the size of $\mathit{unfold}(\lambda v.N_3, \rho'')$. By induction, the cost for the subtrees for self-interpretation are at most $5k_1 + 2$, $5k_2 + 2$ and $5k_3 + 2$. The tree uses (*BETA*) twice, once for $\lambda z.z$, which has size 2 and once for $\mathit{unfold}(\lambda v.\overline{N}_3, \overline{\rho''})$, which is of size at most $4(s-1)$, so the total cost is bounded by $5k_1 + 5k_2 + 5k_3 + 8 + 4(s-1) \leq 5(k_1 + k_2 + k_3 + s) + 4 - s$. Since the smallest possible value for s is 2, we have what we want. □

Combined with the initial cost of $7|M| + 8$, we get

Theorem 14. *If a closed term* M *evaluates by call-by-value to a WHNF in cost* c (*using the substitution-based cost measure*), *selfint*$\lceil M \rceil$ *evaluates by call-by-value to a WHNF in cost at most* $5c + 7|M| + 10$.

4.2. Environment-based cost

The environment-based cost measure is the same as for call-by-name. It is easy to see that the cost of the initial section of the tree is 9. For the rest, the lemma

Lemma 15. *If call-by-value derivation of* $\rho \vdash N \Rightarrow (\lambda y.W,\ \rho')$ *has cost* c (*using the environment-based cost measure*), *then call-by-value derivation of* $\bar{\rho} \vdash \overline{N} \Rightarrow (\lambda y.\overline{W},\ \bar{\rho}')$ *has cost at most* $7c$.

is used. We prove this as before

$N = x$: Both normal evaluation and self-interpretation has cost 1.

$N = \lambda y.W$: The normal evaluation tree has cost 1 while self-interpretation uses (*VAR*) twice and the (*BETA*) rule once for the term $\lambda z.z$, where the expanded environment is of size 2, giving a total cost of 4. This is well below the limit.

$N = N_1\ N_2$: Assuming the subtrees have costs k_1, k_2 and k_3 respectively, the total cost of normal evaluation is $k_1 + k_2 + k_3 + |\rho''| + 1$. By induction, the cost for the subtrees for self-interpretation are at most $7k_1$, $7k_2$ and $7k_3$. The tree uses (*VAR*) and (*BETA*) twice each, the latter once for $\lambda z.z$ (cost 2) and once for $(\lambda v.\overline{N}_3, \overline{\rho''})$, which has cost $|\overline{\rho''}| + 1 = |\rho''| + 3$, so the total cost is bounded by $7k_1 + 7k_2 + 7k_3 + |\rho''| + 7 \leq 7(k_1 + k_2 + k_3 + |\rho''| + 1)$, which is what we want.

Combined with the initial cost of 9, we get

Theorem 16. *If a closed term* M *evaluates by call-by-value to a WHNF in cost* c (*using the environment-based cost measure*), *selfint* $\lceil M \rceil$ *evaluates by call-by-value to a WHNF in cost at most* $7c + 9$.

$$\rho \vdash \lambda x.M \Rightarrow (\lambda x.M,\ \rho) \quad \textit{(LAMBDA)}$$

$$\rho \vdash x \Rightarrow \rho(x) \quad \textit{(VARV)}$$

$$\frac{\rho \vdash M \Rightarrow (\lambda x.M',\ \rho') \quad \rho'[x \mapsto \bullet] \vdash M' \Rightarrow W}{\rho \vdash M\ N \Rightarrow W} \quad \textit{(DUMMY)}$$

$$\frac{\rho \vdash M \Rightarrow (\lambda x.M',\ \rho') \quad \rho \vdash N \Rightarrow V \quad \rho'[x \mapsto V] \vdash M' \Rightarrow W}{\rho \vdash M\ N \Rightarrow W} \quad \textit{(BETAV)}$$

Figure 7. Inference rules for call-by-need evaluation.

5. Call-by-need reduction

Describing call-by-need reduction by a set of inference rules is not as easy as for call-by-name or call-by-value. Typically, a store is threaded through the evaluation and used for updating closures. This is, however, rather complex, so we take a different route: We make the semantics nondeterministic by having two application rules: One that evaluates the argument as for call-by-value, and one that doesn't but inserts a dummy value $\bullet$ in the environment in place of the value of the argument. There is no rule that allows $\bullet$ in computations, so choosing the (*DUMMY*) application rule will lead to a stuck situation if the value of the argument is needed.

The rules in figure 7 model both call-by-need, call-by-value and everything in-between, depending on how often the (*DUMMY*) rule is chosen: If it is never chosen, we get call-by-value and if it is chosen whenever possible, we get call-by-need.

We can define a partial order on inference trees for the same expression by saying that a tree T_1 is less than a tree T_2 if T_2 uses the (*BETAV*) rule whenever T_1 does. For any term, the least (in this ordering) inference tree that computes a non-$\bullet$ result, corresponds to call-by-need reduction of that term to WHNF.

By using an external rule to select from a set of possible inference trees, we have moved parts of the operational behaviour of the language to the meta-level, rather than in the semantic inference rules themselves.

This characterisation of call-by-need may not seem very operational. However, a process that builds a minimal evaluation tree can mimic traditional implementations of call-by-need: When an application is evaluated, the (*DUMMY*) rules is initially used and a $\bullet$ is inserted in the environment. If use of a $\bullet$ is attempted (which will happen if the argument is in fact needed), the origin of the $\bullet$ is traced back to the offending use of the (*DUMMY*) rule. This part of the inference tree is then forcibly overwritten with an use of the (*BETAV*) rule, and the sub-tree for the argument evaluation is constructed. When this is done, the $\bullet$ in the environment is replaced by the correct value[3] and computation is resumed at the place it was aborted. Hence, $\bullet$'s play the rôle of suspensions (thunks) and the replacement of a use

of the (*DUMMY*) rule by a use of the (*BETAV*) rule corresponds to evaluating and updating the suspension.

5.1. Self-interpretation

The initial part of self-interpretation for call-by-need is the same as for the call-by-value case, except that for simple terms, a or b may not be needed and can hence be bound to $\bullet$ and the corresponding evaluations of their closures not occur. However, the cost of the initial portion will (by any reasonable cost measure) be no more than the cost of the call-by-value tree. We will use the same initial environments as for the call-by-value case, but extend the definition of $\bar{\rho}$ to handle variables that are bound to $\bullet$.

$$\begin{array}{ll}\overline{[]} & = \rho_3 \\ \overline{\rho[x \mapsto (\lambda x.P, \rho')]} & = \bar{\rho}[x \mapsto (\lambda x.\overline{P}, \overline{\rho'})] \\ \overline{\rho[x \mapsto \bullet]} & = \bar{\rho}[x \mapsto \bullet]\end{array}$$

Like in the previous cases, we define a call-by-need simulation lemma:

Lemma 17. *If we, using the call-by-need inference rules, can derive* $\rho \vdash N \Rightarrow (\lambda y.W,\ \rho')$ *then we can also derive* $\bar{\rho} \vdash \overline{N} \Rightarrow (\lambda y.\overline{W},\ \overline{\rho'})$.

We prove Lemma 17 in figure 8.

Since Lemma 17 includes the cases where variables in the environment are bound to $\bullet$, we conclude that, if normal evaluation does not need the value of a variable, then neither does the self-interpreter.

We will in the proofs of linear-time self-interpretation also refer to the proofs for the call-by-value case except for the (*DUMMY*) case, as we use the same cost measures and the same constant factors.

We start by counting beta reductions. Our theorem is

Theorem 18. *If a closed term M via the call-by-need semantics evaluates to a WHNF using n call-by-need beta reductions, selfint $\lceil M \rceil$ evaluates to a WHNF using at most $4n+4$ call-by-need beta reductions.*

The corresponding lemma proves simulation using $4n + 1$ steps, after the initial portion. We use the proof for Lemma 10 with the addition of a case for the (*DUMMY*) rule: Normal evaluation uses $k_1 + k_3 + 1$ beta reductions, where k_1 and k_3 are the numbers of beta reductions required for N_1 and N_3. By induction, interpreting $\overline{N_1}$ and $\overline{N_3}$ costs at most $4k_1 + 1$ and $4k_2 + 1$. Additionally, 2 beta reductions are used, so the total cost is bounded by $4(k_1 + k_2 + 1)$, which is one less than our limit.

We can now go on to substitution-based cost. We assign the same cost to the (*DUMMY*) rule as to the (*BETAV*) rule: The size of the extended environment.

Proof.
We prove lemma 17, as per usual, by induction over the structure of the evaluation tree with N at its root. Only the case for the *(DUMMY)* rule differs from the proof of lemma 9, so we omit the other parts.

$N = N_1\ N_2$: Using the *(DUMMY)* rule, the normal evaluation tree is

$$\frac{\dfrac{\cdots}{\rho \vdash N_1 \Rightarrow (\lambda v.N_3, \rho'')} \quad \dfrac{\cdots}{\rho''[v \mapsto \bullet] \vdash N_3 \Rightarrow (\lambda y.W, \rho')}}{\rho \vdash N_1\ N_2 \Rightarrow (\lambda y.W, \rho')}$$

Which (by induction) leads us to the following self-interpretation tree

$$\frac{(*) \quad \dfrac{\cdots}{\overline{\rho''}[v \mapsto \bullet] \vdash \overline{N_3} \Rightarrow (\lambda y.\overline{W}, \overline{\rho'})}}{\overline{\rho} \vdash a\ \overline{N_1}\ \overline{N_2} \Rightarrow (\lambda y.\overline{W}, \overline{\rho'})}$$

where $(*)$ is the tree

$$\frac{\overline{\rho} \vdash a \Rightarrow (\lambda z.z, \rho_1) \quad \dfrac{\cdots}{\overline{\rho} \vdash \overline{N_1} \Rightarrow (\lambda v.\overline{N_3}, \overline{\rho''})} \quad (**)}{\overline{\rho} \vdash a\ \overline{N_1} \Rightarrow (\lambda v.\overline{N_3}, \overline{\rho''})}$$

and $(**)$ is $\quad \rho_1[z \mapsto (\lambda v.\overline{N_3}, \overline{\rho''})] \vdash z \Rightarrow (\lambda v.\overline{N_3}, \overline{\rho''})$.

□

Figure 8. Proof of Lemma 17.

We extend the definition of *unfold* to handle $\bullet$:

$$\mathit{unfold}(P, \rho[x \mapsto \bullet]) \;=\; \mathit{unfold}(P, \rho)[x \backslash d]$$

where d is a free variable that does not occur anywhere else. It is easy to see that the same size limit as before applies: $|\mathit{unfold}(\lambda y.\overline{P}, \bar{\rho})| \leq 4|\mathit{unfold}(P, \rho)|$. Hence, we shall go directly to the theorem

Theorem 19. *If a closed term M evaluates by call-by-need to a WHNF in cost c (using the substitution-based cost measure), selfint $\lceil M \rceil$ evaluates by call-by-need to a WHNF in cost at most $5c + 7|M| + 10$.*

Again, we only state the case for the (*DUMMY*) rule and refer to Lemma 13 for the rest: If normal evaluation has cost k_1 and k_3 for evaluation of N_1 and N_3, the total cost is $k_1 + k_2 + s$, where s is the size of $\mathit{unfold}(\lambda v.N_3, \rho'')$. For self-interpretation, interpretation of $\overline{N_1}$ and $\overline{N_3}$ have by induction costs bounded by $5k_1 + 2$ and $5k_3 + 2$. Additionally, we use (*BETAV*) once

at cost 2 and (*DUMMY*) once at cost $|unfold(\lambda v.\overline{N_3}, \overline{\rho''})| \leq 4|unfold(N_3, \rho'')| = 4(s-1)$. This gives a total cost bounded by $5(k_1 + k_2 + s) - s + 2$, which is well within our limit.

Environment-based cost is no bigger problem:

Theorem 20. *If a closed term M evaluates by call-by-need to a WHNF in cost c (using the environment-based cost measure), selfint $\ulcorner M \urcorner$ evaluates by call-by-need to a WHNF in cost at most $7c + 9$.*

Again, we refer to the proof for the call-by-value case except for an additional case for the proof of Lemma 15 to handle the (*DUMMY*) rule:

Normal evaluation uses the (*DUMMY*) rule at cost $|\rho''|+1$ plus the costs of evaluating N_1 and N_3, which we set at k_1 and k_2. Self-interpretation uses at most $7k_1$ and $7k_3$ to interpret $\overline{N_1}$ and $\overline{N_3}$. To this we add two uses of (*VARV*), one use of (*BETAV*) at cost 2 and the use of (*DUMMY*) at cost $|\overline{\rho''}| + 1 = |\rho''| + 3$. This adds up to $7(k_1 + k_3 + |\rho''| + 1) - 6|\rho''|$, which is within our budget.

6. Conclusion and future work

We have proven that a simple self-interpreter for the pure lambda calculus can do self-interpretation in linear time, i.e. constant overhead. We proved this for reduction to weak head normal form using call-by-name, call-by-value and call-by-need using three different cost measures.

It would be interesting to extend the present work to include studies of self-interpretation cost for reduction to head normal form and full normal form. The author expects these to have linear-time self-interpretation too, but is not currently working on proving this.

Apart from being interesting in its own right, the result is a step towards proving the existence of a linear-time complexity hierarchy for the pure lambda calculus, along the lines of Jones' result for first-order functional and imperative languages [5]. The proof involves a self-interpreter that not only has constant overhead but also counts the amount of time (by some cost measure) it uses. If it can not finish within a set budget of time, the self-interpreter stops with a special error-value. This self-interpreter is then used in a diagonalisation proof reminiscent of the classical halting-problem proof to show that a certain problem can be solved in time kn but not in time n, where k is the interpretation overhead.

We are currently working on this and have sketched a proof for call-by-name reduction to WHNF. However, due to the resource counting the proof is about an order of magnitude harder than the proofs shown in this paper, so we are investigating ways to simplify the proofs.

This study has some relation to the work by Rose [13] on showing that there exists a linear-time hierarchy for CAM, an abstract machine used for implementing higher-order functional languages. This was proven by showing linear-time interpretations between CAM and the language used in Jones' paper. This method does not carry over to the lambda calculus, as such interpretations are not likely to exist, at least not for natural complexity measures for reduction in the lambda calculus.

Rose [12] goes on to attempt to characterise necessary conditions for the existence of a linear-time hierarchy. She states that to support a linear-time hierarchy, a language may not allow constant-time access to a non-constant number of variables, locations, symbols or functions, where the constant is uniform over all programs. The reasoning is that constant-time access to an unbounded number of entities will allow tricks similar to the alphabet-extension used in the speed-up theorem for Turing machines. This would imply that any cost measure for the lambda calculus that allows constant time access to variables (e.g., counting beta-reductions) contradicts the existence of a linear-time hierarchy. However, the proof sketch mentioned above indicates that one such probably does exist. Rose's characterisation is based on the assumption that complexity measures are realistic with respect to real machines, which may be why it doesn't apply to the lambda calculus (the cost measure used in the proof sketch mentioned above is the number of beta reductions, which is hardly realistic).

In [7], a different representation of lambda terms was used. It was based on higher-order abstract syntax, but used a standard-style representation where recursion over the syntax is not encoded in the term itself (see Appendix A). Hence, the self-interpreter needed to use an explicitly coded fixed-point combinator, making it somewhat more complex than the one used in this paper. Redoing the proofs in this paper for that self-interpreter will be much more work due to the larger size, but the same principles should apply and we expect a constant (but much larger) overhead for this case as well.

The use of a nondeterministic operational semantics to encode call-by-need reduction made the proofs for this very simple. In our knowledge, this technique hasn't been used earlier, though a similar notion (replacing a term by $\bullet$) has been used to define neededness [2]. We expect it to be useful for proving other properties about call-by-need reduction.

Our discussion of different cost measures may seem similar to the discussions by, e.g., Frandsen and Sturticant [3] or Lawall and Mairson [6] on cost models for the lambda calculus. However, their cost models are meant to be independent of any particular implementation or reduction order whereas the measures presented here try to mimic specific implementation methods for fixed reduction orders. Likewise, we only address reduction to WHNF while Lawall and Mairson cover reduction to full normal form. Rose [13] cites Lawall and Mairson and argues that their conclusion that there is no single reasonable cost measure for the lambda calculus means that it makes no sense to talk about a linear-time complexity hierarchy. However, by fixing reduction order we *can* find reasonable cost measures and hence get around this problem.

Appendix A: Operations on representations

We first recall the Church-style higher-order representation used for the self-interpreter in this paper:

$$
\begin{array}{ll}
\ulcorner M \urcorner & \equiv \lambda a\, b.\, \overline{M} \\
\quad \textit{where} & \\
\bar{x} & \equiv x \\
\overline{P\,Q} & \equiv a\,\overline{P}\,\overline{Q} \\
\overline{\lambda x.P} & \equiv b\lambda x.\overline{P}
\end{array}
$$

We will assume terms are closed.

In [7], a standard-style higher order representation was used:

$$\begin{array}{ll} \lfloor x \rfloor & \equiv \lambda a\, b\, c.a\, x \\ \lfloor P\, Q \rfloor & \equiv \lambda a\, b\, c.b \lfloor P \rfloor \lfloor Q \rfloor \\ \lfloor \lambda x.P \rfloor & \equiv \lambda a\, b\, c.c\, (\lambda x. \lfloor P \rfloor) \end{array}$$

We can convert between these representations inside the lambda calculus. For example, $C2S \ulcorner M \urcorner \twoheadrightarrow \lfloor M \rfloor$ where $C2S$ is defined by

$$\begin{array}{ll} C2S & \equiv \lambda m.m\, A\, B \\ \textit{where} & \\ A & \equiv \lambda p\, q.\lambda a\, b\, c.b\, p\, q \\ B & \equiv \lambda f.\lambda a\, b\, c.c\, (\lambda x.\, f\, (\lambda a\, b\, c.a\, x), \end{array}$$

We can eliminate the higher-order aspect of the abstract syntax by using numbers to represent variables. If we use the standard representation of numbers:[4]

$$\begin{array}{ll} 0 & \equiv \lambda x\, y.x \\ i+1 & \equiv \lambda x\, y.y\, i \end{array}$$

we can define a first-order, standard-style representation of lambda terms using numbered variables by

$$\begin{array}{ll} \langle x_i \rangle & \equiv \lambda a\, b\, c.a\, i \\ \langle P\, Q \rangle & \equiv \lambda a\, b\, c.b \langle P \rangle \langle Q \rangle \\ \langle \lambda x_i.P \rangle & \equiv \lambda a\, b\, c.c\, i\, \langle P \rangle \end{array}$$

We can, for example, convert from the Church-style representation to the first-order standard-style representation by $C2F \ulcorner M \urcorner \twoheadrightarrow \langle M \rangle$ where $C2F$ is defined by

$$\begin{array}{ll} C2F & \equiv \lambda m.m\, A\, B\, 0 \\ \textit{where} & \\ A & \equiv \lambda p\, q\, i.\lambda a\, b\, c.b\, (p\, i)\, (q\, i) \\ B & \equiv \lambda f\, i.\lambda a\, b\, c.c\, i\, (\, f\, (\lambda j.\lambda a\, b\, c.a\, i)\, (\lambda x\, y.y\, i)) \end{array}$$

Equality predicates are usually easier to code for standard-style representations than for Church-style representations. For example, equality on standard-style numbers m and n can be tested by $eq\ m\ n$ which reduces to $T \equiv \lambda t\ f.t$ if m and n are equal and to $F \equiv \lambda t\ f.f$ if they are not. In the definition of eq below, we use a fixed-point combinator Y.

$$eq \equiv Y\, (\lambda e.\lambda i\ j.i\, (j\, T (\lambda l.F))\, (\lambda k.j\, F\, (\lambda l.e\, k\, l))$$

We can use this equality predicate in our alpha-equivalence tester. For convenience, we let the first argument to the alpha-equivalence tester be in higher-order standard-style

representation and the other argument in first-order standard style representation. In other words, $\alpha\lfloor M\rfloor\langle N\rangle$ reduces to T if $M \equiv N$ and to F if not.

$$
\begin{array}{lcl}
\alpha & \equiv & Y(\lambda a.\lambda m\, n.m\ A\ B\ C) \\
\textit{where} & & \\
A & \equiv & \lambda i.n(\lambda j.eq\, i\ j\ (\lambda r\, s.F)\ (\lambda k\, g.F) \\
B & \equiv & \lambda p\, q.n\ (\lambda j.F)\ (\lambda r\, s.(a\ p\, r)\ (a\, q\, s)F)\ (\lambda k\, g.F) \\
C & \equiv & \lambda f.n(\lambda j.F)\ (\lambda r\, s.F)\ (\lambda k\, g.a\ (f\ k)g)
\end{array}
$$

By composing α with $C2S$ and $C2F$ we obtain an alpha-equivalence tester for the Church-style representation: $\alpha(C2S\lceil M\rceil)\ (C2F\ \lceil N\rceil)$ reduces to T if and only if $M \equiv N$.

Notes

1. We do not consider eta-equivalence, as this is not a valid equivalence for reduction to WHNF.
2. This strategy has a risk of doing unnecessary work, as not all of a functions body may be needed. Hence, most implementations of graph reduction postpone some of the copying until it is known if it is required.
3. This can be done in constant time by overwriting a shared indirection node.
4. Also called "Scott numerals". We use a slightly different representation than the standard numerals in [1].

References

1. Barendregt, H.P. *The Lambda Calculus, its Syntax and Semantics*, 2 edition, North-Holland, Amsterdam, New York, Oxford, 1984. Studies in Logic and the Foundations of Mathematics, vol. 103.
2. Durand, I. and Middeldorp, A. Decidable call by need computations in term rewriting. In *CADE '97*. Springer-Verlag, 1997, pp. 4–18. Lecture Notes in Artificial Intelligence 1249.
3. Frandsen, G. and Sturtivant, C. In *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture*, John Hughes (Ed.). Springer-Verlag, Cambridge, Massachusetts, 1991, pp. 289–312. Lecture Notes in Computer Science 523.
4. Hennessy, M. *The Semantics of Programming Languages*. John Wiley and Sons, Chichester, New York, Brisbane, Toronto, Signapore, 1990.
5. Jones, N.D. Constant time factors *do* matter. In *STOC '93. Symposium on Theory of Computing*, S. Homer (Ed.). ACM Press, 1993, pp. 602–611.
6. Lawall, J.L. and Mairson, H.G. Optimality and inefficiency: What isn't a cost model of the lambda calculus. In *Proceedings of ICFP '95*, R.K. Dybvig (Ed.). ACM, ACM Press, 1996, pp. 92–101.
7. Mogensen, T.Æ. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming* **2**(3) (1992) 345–364.
8. Mogensen, T.Æ. Self-applicable partial evaluation for pure lambda calculus. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, C. Consel (Ed.). ACM, Yale University, 1992, pp. 116–121.
9. Mogensen, T.Æ. Self-applicable online partial evaluation of the pure lambda calculus. In *Proceedings of PEPM '95*, W.L. Scherlis (Ed.). ACM, ACM Press, 1995, pp. 39–44.
10. Pfenning, F. and Elliot, C. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, ACM, ACM Press, 1988, pp. 199–208.
11. Plotkin, G.D. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Aarhus University, Denmark, Sept. 1981.
12. Rose, E. Characterizing computation models with a constant factor time hierarchy. In *DIMACS Workshop On Computational Complexity and Programming Languages*, B. Kapron (Ed.). New Jersey, USA, July 1996. DIMACS, RUTCOR, Rutgers University.
13. Rose, E. Linear time hierarchies for a functional language machine model. *Science of Computer Programming* **32**(1–3) (1998) 109–143.