Deriving Abstract Machines with Sharing

Kristoffer H. Rose
DIKU, University of Copenhagen*

Submitted to ICFP '96, October 1995

Abstract

It is known that abstract machines can be derived from calculi with explicit substitution or environments (Hannan and Miller 1990, Curien 1991, Abadi, Cardelli, Curien and Lévy 1991). In this paper we first show how an abstract machine can be derived from a λ-calculus with explicit substitution using names. We then show how we can enrich this with *sharing information* which then ripples through the derivation, to reappear in the abstract machine as a form of 'addressing' very similar to that used in graph reduction for functional language implementations.

Key words: lambda calculus, functional programming, explicit substitution, computational models, abstract machines, sharing, graph reduction.

1 Introduction

In the sixties our community searched for *computational models* suitable for expressing the complexity of solving recursive systems of equations, the core of today's functional programming languages. The breaktrough was the SECD machine of Landin (1964), and since then the notion of *abstract machine* has been a key notion in studying complexity of functional programs.

The penalty, however, is that the evaluation order is fixed *before* anything is known about the complexity, so it is difficult to reason about the relative complexity of different reduction strategies, in particular the 'improvement' of program transformations, which typically involve reductions that are not allowed inside the system itself.

One solution is to use *explicit substitution* calculi with elementary reduction steps yet without a predetermined reduction strategy. Instead such calculi should be confluent: this makes it possible to first design a calculus where the reduction length defines a complexity measure, and *then* derive an abstract machine for a particular strategy. The seminal λσ-calculus of Abadi et al. (1991) developed from Curien's (1986) Categorical Combinatory Logic was the first to make this possible (even though the complexity issue is not raised).

Independently of this, research into implementation of functional programming languages combined the notion of an abstract machine with *sharing* by keeping track of different instances of the same computations. The most well-known result of this development is the 'G-machine' of Johnsson (1984) which realises the supercombinator reduction of Hughes (1982) in a complexity-preserving way.

*E-mail: *kris@diku.dk*. World Wide Web URL: *http://www.diku.dk/~kris/*. Address: Datalogisk Institut ved Københavns Universitet, Universitetsparken 1, DK–2100 København Ø, Denmark. Fax: (+45) 35321401.

It seems an obvious route to attempt to recombine the above ideas, reflecting practice back into theory, and obtaining abstract machines that include sharing. However, this requires a notion of sharing in the *source* language, in our case the λ-calculus. Such have been studied in several instances since Wadsworth (1971) where the notion of 'origin labelling' was introduced, in fact most work in this area has since then been based on the generalised labelling of Lévy (1978); we mention some in the paper.

Our aim is thus to attempt to unite three areas: functional programming language implementation techniques, functional program complexity as manifested by explicit substitution, and λ-calculi with sharing. In section 2 we recall a simple explicit substitution calculus enriched with explicit garbage collection, λxgc, and we show that it is a conservative extension of the λβ-calculus. It has two distinguishing features that we shall see makes it very practical for our purpose. One is the use of names instead of some encoding à la de Bruijn (1972), which makes it possible to reason about memory addresses in a free way. The other is the presence of garbage collection, which makes it possible to reason about 'real' space behaviour (since functional language implementations can reclaim space occupied by 'garbage'). In section 3 we show a 'traditional' derivation of an abstract machine as it appears when applied to λxgc. Finally, in section 4 we introduce a variant of the λxgc-calculus with *addresses* and show how these filter through the derivation and lead to a notion of 'memory management' in the resulting abstract machine.

The reader is expected to be familiar with basic notation and concepts of λ-calculus (Barendregt 1984) and term rewriting (Klop 1992).

2 An Explicit Substitution Calculus with Names and Garbage Collection

This section is about the λxgc-calculus: an *explicit substitution* calculus in the tradition of λσ (Abadi et al. 1991). However, we have chosen to use *variable names* instead of indices à la de Bruijn (1972). This makes the calculus simpler by making it possible to use only 'naïve direct substitution' following Rose (1993a). Furthermore, λxgc shares with λs of Kamareddine and Ríos (1995) and λυ of Lescanne (1994) that there is no syntax for explicit composition of substitutions. Finally, the calculus has explicit *garbage collection* since it is useful and fairly easy to specify using names.

We first present the λx-terms we will use throughout the paper, and generalise the standard related concepts from the λ-calculus to it. Then we present λxgc-reduction, and finally we summarise some properties of λxgc that are important to the later developments.

2.1 Definition (λx-terms). The set of λx-terms Λx is ranged over by the letters MNPQ and R, and defined inductively by $M ::= x \mid \lambda x.M \mid MM \mid M\langle x := M\rangle$ where the letters $xyzvw$ range over an infinite set of variables. As usual we use parentheses to indicate subterm structure and write $\lambda xy.M$ for $\lambda x.(\lambda y.M)$ and MNP for $(MN)P$; explicit substitution is given highest precedence so $\lambda x.MNP\langle y := Q\rangle$ is $\lambda x.((MN)(P\langle y := Q\rangle))$.

The λx-terms include as a subset the ordinary λ-terms, Λ; we will say that a λx-term is 'pure' if it is also a λ-term, *i.e.*, if it has no subterms of the form $M\langle x := N\rangle$. The usual β-reduction is denoted $\xrightarrow[\beta]{}$ (and only defined on pure terms, of course). Several other standard concepts generalise naturally:

2.2 Definitions (λ-term concepts).

a. The *free variable set* of a λx-term M is denoted $\mathrm{fv}(M)$ and defined inductively over M as for λ-calculus plus $\mathrm{fv}(M\langle x := N\rangle) = (\mathrm{fv}(M) \setminus \{x\}) \cup \mathrm{fv}(N)$. A λx-term M is *closed* iff $\mathrm{fv}(M) = \varnothing$.

Substitution generation: $\xrightarrow[b]{}$ is the contextual closure (modulo $\equiv$) of

$$(\lambda x.M)N \to M\langle x := N\rangle \qquad \text{(b)}$$

Explicit substitution: $\xrightarrow[x]{}$ is defined as the contextual closure (modulo $\equiv$) of the union of

$$x\langle x := N\rangle \to N \qquad \text{(xv)}$$
$$x\langle y := N\rangle \to x \quad \text{if } x \not\equiv y \qquad \text{(xvgc)}$$
$$(\lambda x.M)\langle y := N\rangle \to \lambda x.M\langle y := N\rangle \qquad \text{(xab)}$$
$$(M_1M_2)\langle y := N\rangle \to M_1\langle y := N\rangle\ M_2\langle y := N\rangle \qquad \text{(xap)}$$

Note: 2.3 ensures that in (xab) there is neither variable capture ($x \notin fv(N)$) nor clash ($x \not\equiv y$).

Garbage collection: $\xrightarrow[gc]{}$ is the contextual closure (modulo $\equiv$) of

$$M\langle x := N\rangle \to M \quad \text{if } x \notin fv(M) \qquad \text{(gc)}$$

The subterm N in $M\langle x := N\rangle$ is called *garbage* if $x \notin fv(M)$.

Figure 1: λxgc-reduction step

b. The result of *renaming all free occurrences of* y *in* M *to* z is written $M[y := z]$ and defined inductively over M as for λ-calculus plus $(M\langle x := N\rangle)[y := z] = M[x := x'][y := z]\ \langle x' := N[y := z]\rangle$ with $x' \notin fv(\lambda x.M) \cup \{y, z\}$.

c. That two terms are *α-equivalent* is written $M \equiv N$. It is as for λ-calculus plus $M\langle x := N\rangle \equiv P\langle y := Q\rangle$ if $N \equiv Q$ and $M[x := z] \equiv P[y := z]$ for $z \notin fv(MP)$.

As usual we will work modulo α-equivalence, using $\equiv$ for identity of terms. In order to avoid trivial conflicts of variable names we will use the convention of Barendregt (1984, 2.1.13).

2.3 Convention (bound variable naming). When a group of terms occurs in a mathematical context (like a definition, proof, etc.), then all variables bound in these terms are chosen to be distinct and different from all free variables in them.

Now we are ready to define λxgc-reduction:

2.4 Definitions (λxgc-reduction step). $\xrightarrow[bxgc]{}$ is the union of all reductions defined in figure 1; $\xrightarrow[xgc]{}$ is the union $\xrightarrow[x]{} \cup \xrightarrow[gc]{}$.

2.5 Remark. The above reduction definition is based directly on the usual definition of β-reduction with substitution by writing it explicitly, *i.e.*, changing the implicit definition of 'meta-substitution' to an explicit syntactic substitution without changing anything else. In addition we have just added the definition of garbage collection directly, thus introducing additional overlap between the rules.

Next we summarise some properties of the subrelations that are essential when relating this calculus to the pure λ-calculus, in particular when explaining the relationship between explicit substitution, garbage collection, and traditional (implicit) substitution. Notice that the properties we state for λxgc-reduction and $\xrightarrow[xgc]{}$ will generally imply the same for λx-reduction and $\xrightarrow[x]{}$ (which are the same but omitting (gc)).

2.6 Propositions. a. $\underset{b}{\to}$SN, b. $\underset{b}{\to}\Diamond$, c. $\underset{xgc}{\to}$SN, and d. $\underset{xgc}{\to}$CR.

Proof. The only nontrivial property is 2.6c: $\underset{xgc}{\to}$SN is shown by finding a map $h : \Lambda x \to \mathbb{N}$ such that for all $M \underset{x}{\to} N$: $h(M) > h(N)$. This is easily verified for the map defined inductively by $h(x) = 1$, $h(MN) = h(M) + h(N) + 1$, $h(\lambda x.M) = h(M) + 1$, and $h(M\langle x := N\rangle) = h(M) \times (h(N) + 1)$. □

2.7 Notation (normal forms). $\downarrow(M)$ is the $\to$-nf of M and $\twoheadrightarrow$ is 'the restriction of $\twoheadrightarrow$ to reductions to normal form' (thus $\twoheadrightarrow \subseteq \twoheadrightarrow$). In particular, $\downarrow_x(M)$ is the $\underset{x}{\to}$-nf of M where we say M is *pure* if $M \equiv \downarrow_x(M)$ (or equivalently $M \twoheadrightarrow M$); and $\downarrow_{gc}(M)$ is the $\underset{gc}{\to}$-nf of M.

Clearly this notion of a 'pure' term corresponds to the informal use of the concept above as 'ordinary λ-term'. With this we are able to establish the relation between explicit and pure substitution: we show that the behaviour of the substitutions $P\langle x := Q\rangle$ relates well to the usual meta-substitution $P[x := Q]$.

2.8 Lemma (representation). For all terms M, N and variable x, $\downarrow_x(M\langle x := N\rangle) \equiv \downarrow_x(M)[x := \downarrow_x(N)]$ where $P[x := Q]$ denotes the term obtained by (usual) substitution of the (pure) term Q for all free occurrences of x in (the pure term) P as a meta-operation. In particular $\downarrow_x(M\langle x := N\rangle) \equiv M[x := N]$ for pure M and N.

Proof. Induction on the number of symbols in the non-substitution subterms. □

Now that we have established the properties of the substitution subrelation, we can compare to the ordinary (pure) β-reduction. We show that $\underset{bxgc}{\to}$ is a conservative extension of $\underset{\beta}{\to}$, and thus confluent.

2.9 Lemma (projection). For all M, $\begin{array}{ccc} M & \xrightarrow{\ b\ } & N \\ {\scriptstyle x}\downarrow\!\!\downarrow & & {\scriptstyle x}\downarrow\!\!\downarrow \\ \downarrow_x(M) & \underset{\beta}{\twoheadrightarrow} & \downarrow_x(N) \end{array}$.

Proof sketch. Induction on the structure of M, using 2.8. □

An interesting consequence of representation and projection is the following:

2.10 Corollary (substitution lemma). $M\langle x := N\rangle\langle y := P\rangle \underset{xgc}{\twoheadleftarrow\!\!\twoheadrightarrow} M\langle y := P\rangle\langle x := N\langle y := P\rangle\rangle$

For the pure calculus this conversion[1] degenerates to an identity which is the usual substitution lemma.

The above suffices to show that λxgc is indeed a conservative extension of the $\lambda\beta$-calculus:

2.11 Theorem. *For pure terms M, N: $M \underset{bxgc}{\twoheadrightarrow} N$ iff $M \underset{\beta}{\twoheadrightarrow} N$.*

Proof. Induction on length of $\underset{\beta}{\twoheadrightarrow}$-reduction for 'if' (using representation) and of $\underset{bxgc}{\twoheadrightarrow}$-reduction for 'only if' (using projection). □

An easy consequence of all this is that 'modular' properties of $\underset{\beta}{\to}$ are preserved for $\underset{bxgc}{\to}$, in particular:

2.12 Corollary. $\underset{bxgc}{\to}$CR. Proof. *Standard interpretation diagram.*

Finally we mention

2.13 Theorem (preservation of SN). *A pure term is strongly normalising for β-reduction if and only if it is strongly normalising for $\underset{bxgc}{\to}$-reduction.* Proof *in Bloo and Rose (1995).* □

[1] To avoid overloading $=$ we use $\twoheadleftarrow\!\!\twoheadrightarrow$ for $\to$-conversion, *i.e.*, the smallest equivalence relation containing $\to$.

3 Deriving Abstract Machines

By 'abstract machine' we generally denote reduction systems where each reduction step can be implemented as a simple operation in the sense that it can be executed on a concrete machine (*aka* 'computer') in constant time. This makes abstract machines for a particular calculus useful for studying the running times of programs in the calculus.

In this section we will derive an abstract machine for a restriction of the untyped $\lambda\beta$-calculus that is often claimed as the basis for implementations: Abramsky's (1990) 'lazy' calculus, $\lambda\ell$.[2] We first give the proper definition of the rewrite system which our abstract machine should mimic, then define what an abstract machine is, formally. This makes it possible to outline the *derivation principle* which we then follow until an abstract machine is obtained. The presentation below only uses term rewriting notions even though the precise formulation involves definitions and results from combinatory reduction systems (Klop, van Oostrom and van Raamsdonk 1993).

3.1 Definition (Abramsky's (1990) 'lazy' λ-calculus). $\lambda\ell$-reduction is the relation $\Downarrow_\ell$ inductively defined on closed λ-terms by

$$\lambda x.M \Downarrow_\ell \lambda x.M \qquad \frac{M \Downarrow_\ell \lambda x.P \quad P[x := N] \Downarrow_\ell Q}{MN \Downarrow_\ell Q}$$

Clearly this is a restriction of the $\lambda\beta$-calculus: $\Downarrow_\ell \subset \twoheadrightarrow_\beta$. However, the definition does not readily show the reduction strategy in that it is a 'big step' definition: it only defines reduction to normal form. Or to be precise: the reduction strategy is present only implicitly as a 'proof construction strategy' which is fine except it is not so easy to reason about as a reduction. Hence we will instead use an equivalent 'small step semantics' (called 'structural operational semantics' by Plotkin 1981).

3.2 Definition ($\lambda\ell$-reduction step). $\to_\ell$ is inductively defined by

$$(\lambda x.M)N \to M[x := N] \quad (\beta) \qquad \frac{M \to P}{MN \to PN} \quad (\text{AppL})$$

3.3 Proposition. $\twoheadrightarrow_\ell = \Downarrow_\ell$.

Proof. Immediate by isomorphism between $\twoheadrightarrow_\ell$-reduction sequences and $\Downarrow_\ell$-derivation trees. □

3.4 Remark (evaluation strategy concepts). This description clearly separates the 'calculus' and 'strategy' issues: the calculus is the usual $\lambda\beta$, *i.e.*, the contextual closure of (β), and the strategy is the restriction imposed by only allowing reductions in the particular contexts as described by (AppL). This means that $\to_\ell$ is not contextual, of course; hence we will have to use induction over the depth of derivations rather than on terms. We will follow the tradition of logic and classify all the 'original' rules as *axioms* and the strategy constraints as *inferences*.

3.5 Definitions (abstract machine).

a. A rewrite system is *local* if there is a (globally) constant bound on the amount of new structure created by any single rewrite.

[2] The tag 'lazy' is slightly misleading in this case: $\lambda\ell$ is more precisely described as a calculus with *weak leftmost outermost* reduction: weak because no redex inside an abstraction is ever reduced, and leftmost outermost because the leftmost redex is always reduced when there are several. In some traditions this is stated as reduction of *programs* (the same as closed terms) to *weak head normal form* (whnf) where whnf is defined as "an abstraction or an application where the head is a variable". Clearly the only closed terms in whnf are abstractions, hence the definition is the same! We will see in section 4, however, that these definitions are equivalent to usual lazy evaluation when sharing is included.

b. A rewrite system is *flat* if all redexes are always at the root.

c. An *abstract machine* is a local and flat rewrite system. We call the terms for *configurations*.

Thus there are two problems with the $\underset{\ell}{\to}$-relation: it is not local because of the use of substitution, and it is not flat because of the (AppL) inference! We remedy the first by introducing explicit substitution and the second is captured by the following methodology inspired by the techniques of Hannan and Miller (1990).

3.6 Principle (abstract machine derivation). Go through successive transformations as follows until the system is flat:

"Search for theorem": prove structural property of the system that might be useful in an implementation, typically a preservation property or simply one of the inferences.

"Represent": find a term structure that makes it possible to exploit the proved property in the system, 'internalising' the insight of it (and hence flattening the terms).

"Fold": transform the system by adding reductions using the proper function symbol, such that it uses the new representation thus exploiting the theorem.

"Simplify": remove redundancy.

The advantage is that the structures we need will suggest themselves and additionally each transformation is so mechanic that correctness is immediate.

First locality by introducing explicit substitution. This involves our first design decision: how should the strategy be generalised to terms of the form $M\langle x := N\rangle$? Our choice is reflected in the inference (SubL) of the following definition: it restricts reduction in $M\langle x := N\rangle$ to $\underset{x}{\to}$-reductions in M. This captures delaying the substitution steps – it cannot be any more restrictive because substitutions may not be composed (there is no axiom covering the case $M\langle y := N\rangle\langle z := P\rangle$) hence substitutions have to be resolved innermost-first.

3.7 Definition ($\lambda\ell$xgc-reduction step). $\underset{\ell\mathtt{xgc}}{\longrightarrow}$ combines the axioms (b,xv,xvgc,xab,xap,xgc) of 2.4 (without taking the contextual closure), with the new (strategy) inferences

$$\frac{M \underset{\ell\mathtt{xgc}}{\longrightarrow} P}{MN \underset{\ell\mathtt{xgc}}{\longrightarrow} PN}\ (\mathrm{AppL}) \qquad \frac{M\langle y := N\rangle \underset{\ell\mathtt{xgc}}{\longrightarrow} Q}{M\langle y := N\rangle\langle z := P\rangle \underset{\ell\mathtt{xgc}}{\longrightarrow} Q\langle z := P\rangle}\ (\mathrm{SubL})$$

3.8 Remark. The system is not deterministic: there is overlap between the (gc) axiom and most of the other rules which corresponds to the reality of functional programming that garbage collection can occur at any moment – and also highlights the fact that we do not include garbage collection in our complexity measure. It is not even confluent: $(\lambda x.x)\langle y := \lambda x.x\rangle \underset{\mathtt{gc}}{\longrightarrow} \lambda x.x \not\twoheadrightarrow\!\!\!\!\twoheadleftarrow \lambda x.x\langle y := \lambda x.x\rangle$!

3.9 Proposition. $\underset{\ell\mathtt{xgc}}{\twoheadrightarrow} \cdot \underset{x}{\twoheadrightarrow} = \underset{\ell}{\twoheadrightarrow}$.

Proof. Immediate from 2.11 and the observation that with complete substitution (provided by $\underset{x}{\twoheadrightarrow}$) any normal form of $\underset{\ell}{\to}$ can be reached. □

In the rest of this section we derive an abstract machine using the outlined methodology.

We will first look at properties of the 'redex search principle' used by the lazy strategy. The intuition that it is 'leftmost outermost' is captured by the presence of (AppL):

3.10 Proposition. M closed and $M \underset{\ell\mathtt{xgc}}{\longrightarrow} P$ implies for all closed N that $MN \underset{\ell\mathtt{xgc}}{\longrightarrow} PN$.

Axioms:

$$(MN\ ,\ S) \to (M\ ,\ N \cdot S) \qquad \text{(S)}$$
$$(\lambda x.M\ ,\ N \cdot S) \to (M\langle x := N\rangle\ ,\ S) \qquad \text{(bS)}$$
$$(x\langle x := N\rangle\ ,\ S) \to (N\ ,\ S) \qquad \text{(xvS)}$$
$$(x\langle y := N\rangle\ ,\ S) \to (x\ ,\ S) \quad \text{if } x \not\equiv y \qquad \text{(xvgcS)}$$
$$((\lambda x.M)\langle y := N\rangle\ ,\ S) \to (\lambda x.M\langle y := N\rangle\ ,\ S) \qquad \text{(xabS)}$$
$$((M_1M_2)\langle y := N\rangle\ ,\ S) \to (M_1\langle y := N\rangle\ ,\ M_2\langle y := N\rangle \cdot S) \qquad \text{(xapS)}$$
$$(M\langle y := N\rangle\ ,\ S) \to (M\ ,\ S) \quad \text{if } y \notin \mathrm{fv}(M) \qquad \text{(xgcS)}$$

Inferences:

$$\frac{(M\langle y := N\rangle\ ,\ S) \to (Q\ ,\ S)}{(M\langle y := N\rangle\langle z := P\rangle\ ,\ S) \to (Q\langle z := P\rangle\ ,\ S)} \qquad \text{(SubLS)}$$

Figure 2: Intermediate $\lambda\ell$-machine: $\lambda\ell$xgcS. Representation of closed λ-term M is (M, ϵ).

$$(M\langle x := N\rangle\ ,\ E\ ,\ S) \to (M\ ,\ \langle x := N\rangle E\ ,\ S) \qquad \text{(E)}$$
$$(MN\ ,\ E\ ,\ S) \to (M\ ,\ E\ ,\ NE \cdot S) \qquad \text{(SE)}$$
$$(\lambda x.M\ ,\ \epsilon\ ,\ N \cdot S) \to (M\ ,\ \langle x := N\rangle\ ,\ S) \qquad \text{(bSE)}$$
$$(x\ ,\ \langle x := N\rangle E\ ,\ S) \to (N\ ,\ E\ ,\ S) \qquad \text{(xvSE)}$$
$$(x\ ,\ \langle y := N\rangle E\ ,\ S) \to (x\ ,\ E\ ,\ S) \quad \text{if } x \not\equiv y \qquad \text{(xvgcSE)}$$
$$(\lambda x.M\ ,\ \langle y := N\rangle E\ ,\ S) \to (\lambda x.M\langle y := N\rangle E\ ,\ \epsilon\ ,\ S) \qquad \text{(xabSE)}$$
$$(M\ ,\ \langle y := N\rangle E\ ,\ S) \to (M\ ,\ E\ ,\ S) \quad \text{if } y \notin \mathrm{fv}(M) \qquad \text{(xgcSE)}$$

Figure 3: Abstract $\lambda\ell$-machine: $\lambda\ell$xgcSE. Representation of closed λ-term M is (M, ϵ, ϵ).

The constructive contents of this is that reduction of applications can be 'flattened' if we transform the representation of a term $MN_1 \ldots N_k$ into the pair[3] $(M, N_1 \cdot \ldots \cdot N_k)$ and only reduce on M: then we get the same possible reductions until M becomes an abstraction. We use it by adding the representation as an axiom and fold the system. The resulting first machine approximation is the machine given in figure 2. As might be expected, folding of the rules with the representation has removed (AppL) completely since it would become an identity rewrite. The correctness is immediate:

3.11 Proposition. For closed M, $M \xrightarrow[\ell\mathrm{xgc}]{}\!\!\!\!\twoheadrightarrow \lambda x.N$ iff $(M, \epsilon) \xrightarrow[\ell\mathrm{xgcS}]{}\!\!\!\!\twoheadrightarrow (\lambda x.N, \epsilon)$.

The system is still not flat because of (SubLS). This is easily obtained, however, by using the following similarly simple observation:

3.12 Proposition. $(M\langle x := N\rangle, S) \xrightarrow[\ell\mathrm{xgcS}]{} (Q, S)$ implies $(M\langle x := N\rangle\langle y := P\rangle, S) \xrightarrow[\ell\mathrm{xgcS}]{} (Q\langle y := P\rangle, S)$.

Again we transform the system by adding an axiom corresponding to the representation change $(M\langle x := N\rangle, S)$ to $(M, \langle x := N\rangle, S)$ and fold, noting that the non-substitution rules cannot be invoked

[3] We will use tuples $(\_, \ldots)$ for 'machine configurations'; futhermore we allow two kinds of sequences: simple concatenation (with highest precedence) and lists constructed with infix $\cdot$. The empty sequence is denoted ϵ (both kinds).

in a substitution-context (because 3.12 only holds in a context with at least one substitution). The result is shown in figure 3.

3.13 Proposition. For closed M, $(M, \epsilon, \epsilon) \xrightarrow[\ell\mathbf{xgcS}]{}\!\!\!\!\!\rightarrow (\lambda x.N, \epsilon, \epsilon)$ iff $(M, \epsilon, \epsilon) \xrightarrow[\ell\mathbf{xgcSE}]{}\!\!\!\!\!\rightarrow (\lambda x.N, \epsilon, \epsilon)$.

3.14 Remark (variable convention). Note it still has some side conditions on variables, namely the side condition to (xgcSE) and the side conditions to (xabSE) implicit in the variable convention. The first we keep because in this paper in that we do not consider the complexity of the actual garbage collection. However, we can eliminate the two implicit side conditions of (xabSE): The first, $x \not\equiv y$ (to avoid 'variable clash'), is easy to check explicitly, and we can replace (xabSE) with

$$(\ \lambda x.M\ ,\ \langle x := N\rangle E\ ,\ S\ ) \to (\ \lambda x.M\ ,\ E\ ,\ S\ ) \qquad \text{(xabSE1)}$$

$$(\ \lambda x.M\ ,\ \langle y := N\rangle E\ ,\ S\ ) \to (\ \lambda x.M\langle y := N\rangle\ ,\ E\ ,\ S\ ) \qquad \text{(xabSE2)}$$

where we have also folded with the garbage collection rule in the case $x \equiv y$ since $x \notin \mathrm{fv}(\lambda x.M)$. The second condition, $x \notin \mathrm{fv}(N)$ (to avoid 'variable capture'), is trivially true because of the following:

3.15 Proposition. Assume M pure and closed and $(M, \epsilon, \epsilon) \xrightarrow[\ell\mathbf{xgcSE}]{}\!\!\!\!\!\rightarrow (N, \langle x_1 := N_1\rangle \cdots \langle x_n := N_n\rangle, P_1 \cdot \ldots \cdot P_m)$. Then all of $N\langle x_1 := N_1\rangle \cdots \langle x_n := N_n\rangle$, N_i, and P_i, are closed.

Proof. Easy case analysis of the rules that all preserve the property. □

All the above transformations combine into the following:

3.16 Theorem. *For closed* M, $M \Downarrow_\ell \lambda x.P$ *iff* $(M, \epsilon, \epsilon) \xrightarrow[\ell\mathbf{xgcSE}]{}\!\!\!\!\!\rightarrow (N, \epsilon, \epsilon)$ *and* $\downarrow_x(N) \equiv \lambda x.P$.

However, we cannot really be sure that the machine does the same number of reductions as the λxgc-system since we have allowed duplication of subterms. This is the motivation for incorporating sharing.

4 Explicit Sharing

In this section we demonstrate how explicit sharing can be added to the λxgc-calculus, and show how this ripples through a generic abstract machine derivation (as presented in section 3) and turn up as essentially *graph reduction*.

4.1 Remark. The notion of an address is inherently global, and existing expositions of sharing also make the notion of address a global one, maintaining a 'store' component mapping all addresses to their contents, *e.g.*, both Launchbury (1993) and Sestoft (1994). However, this is not very 'abstract' because the only operations that actual functional programming languages use the store for two things: *allocation* of a new address in the store, and *share* a particular address from two contexts. Thus the actual addresses are not used at all! This is very similar to the concept of *parallel reduction* in rewriting which we will use below.

So how do we add sharing to the abstract machine of section 3? The most obvious way would be to add adresses to the λ-calculus but this is not possible without redefining substitution (as witnessed by the $\hat{}$ operator of Launchbury 1993). We will instead add sharing to the explicit substitution calculus where the locality ensures that the number of fresh addresses we will need in each reduction step is bounded. We will take the viewpoint that addresses contain *sharing information* thus an 'unaddressed' term simply adds no sharing.

4.2 Definition (λxa-terms). Λxa is Λx inductively extended with $M ::= \cdots \mid M^a$ where abc range over an infinite set of 'addresses'. The new form is called an *addressing* where a *addresses* M. *Unravelling* $\wr M \int$ denotes the λx-term obtained by simply removing all the addresses in M; the notions of free variables, renaming, and α-equivalence, are defined using this (hence ignore addresses). Any subterm of an addressing is *shared* and conversely if there is no address in the context of a subterm then it is not shared. Finally, $M[N^a]$ is the *updating* obtained by replacing all subterms P^a in M (if any) by N^a; and (by convention) $M[\epsilon] \equiv M$.

Assuming that a meaningful addressing has been found (we come to this below), then what does a reduction step look like? The obvious generalisation of λxgc-reduction is to ensure that any duplication in the rules is remembered in the form of a *sharing introduction*. A quick glance over the λxgc-rules reveals that the only thing that is ever duplicated is a substitution (namely in (xap)). Thus if all substitutions have an address then there is never loss of sharing! For this we will need the notion of *fresh*[4] for 'not occuring anywhere else'. We also need *sharing elimination* to get rid of addresses when they block reduction (*e.g.*, a should be removed in $(\lambda x.M)^a N$ to allow $\xrightarrow[b]{}$-reduction).

4.3 Definition (λaxgc-reduction step). $\xrightarrow[baxgc]{}$ is a contextual closure defined as λxgc-reduction in 2.4 except (b) is replaced with

$$(\lambda x.M)N \to M\langle x := N^a\rangle \qquad a \text{ fresh} \tag{ba}$$

and we add the sharing-elimination subrelation

$$M^a \to M \tag{a}$$

4.4 Proposition. $M \xrightarrow[baxgc]{} N$ implies $\wr M \int \xrightarrow[bxgc]{\equiv} \wr N \int$. *Proof.* Immediate. □

In practice we wish to restrict attention to terms where the addressing describes meaningful sharing, *i.e.*, where all instances of a particular address has the same subterm beneath it.

4.5 Definition (well-formed addressing). We write 'M wfa' if M has a mapping from addresses to subterms: the *store function* '$M/$' such that $M/a \equiv P^a$ for all subterms P^a of M and $M/a \equiv \epsilon$ elsewhere.

4.6 Remark (graph reduction intuition). Also a term can have many addresses, *e.g.*, $(M^a)^b$. Abstractly this just means that it is 'involved' in several sharing relationships (this is often realised by 'indirection nodes' in graph reduction) . However, the restriction above means that these must be nested properly, *e.g.*, the term $((M^a)^b)(M^b)$ is not wfa. In fact, the idea with wfa is intended to be similar to graph reduction. In particular (a) corresponds to *indirection removal* when it is a subterm of an addressing $(M^a)^b$ since the a address is eliminated in all the places where it is used as a synonym for b (a can occur elsewhere but all bs must addresses M^a by 4.5); it corresponds to *copying* in all other contexts because the removed address then implicitly means that a new unique one is now used.

However λaxgc-reduction does not preserve wfa! We need to use what is usually called *parallel reduction*: simultaneous reduction of all terms that have the same address. This is defined as follows:

4.7 Definition (λaxgc||-reduction). Let $M \xrightarrow[baxgc]{} N$. Consider the contracted redex R:

[4] We could use some encoding with labels à la Wadsworth (1971) which can be used to ensure the same but since it is rather verbose (in ways not important to this presentation) we stay with the simpler, global notion.

Shared: Let R^a be the smallest subterm of N containing or equal to the contracted redex: then $M \xrightarrow[\text{baxgc}]{} N[R^a]$.

Not shared: Then $M \xrightarrow[\text{baxgc}]{} N$.

The following states that the sharing is correct.

4.8 Propositions. a. $\xrightarrow[\text{baxgc}]{} \subset \xrightarrow[\text{baxgc}]{+}$. *Proof.* Immediate because there is no overlap. □

b. M wfa and $M \xrightarrow[\text{baxgc}]{} N$ implies N wfa. *Proof.* Easy induction. □

Notice how 4.8a has as a consequence that *reduction with sharing is at least as efficient as without* (for any reduction strategy).

We now repeat the derivation process of section 3 except for using parallel reduction. Since parallel reduction may involve reduction in any part of a term, the initial system is the following where the essential insight is that we *record the nearest sharing point* and carefully use it to update whenever we have reduced:

4.9 Definition (λℓaxgc-reduction step). $\xrightarrow[\ell\text{axgc}]{a}$ is inductively defined by the axioms of 4.3 augmented to set $a = o$ and with the following axiom added:

$$M^a\langle y := N\rangle \xrightarrow[\ell\text{axgc}]{o} M\langle y := N\rangle \qquad \text{(xa)}$$

and the (strategy) inferences

$$\frac{M \xrightarrow[\ell\text{axgc}]{a} Q}{MN \xrightarrow[\ell\text{axgc}]{a} Q\ N[Q/a]}\ \text{(AppL)} \qquad \frac{M\langle y := N\rangle \xrightarrow[\ell\text{axgc}]{o} Q}{M\langle y := N\rangle\langle z := P\rangle \xrightarrow[\ell\text{axgc}]{o} Q\langle z := P\rangle}\ \text{(SubL)}$$

$$\frac{M \xrightarrow[\ell\text{axgc}]{o} Q}{M^b \xrightarrow[\ell\text{axgc}]{b} Q^b}\ \text{(Share1)} \qquad \frac{M \xrightarrow[\ell\text{axgc}]{a} Q}{M^b \xrightarrow[\ell\text{axgc}]{a} Q^b}\ a \neq o \quad \text{(Share2)}$$

The a annotation on the arrow is called the 'update address'; o is a unique *dummy address*. Notice that substitutions never set an update address.

This time the first property we use is the combination of (AppL) and (Share1,2) since those are the context-free rules. The folding proceeds very much like the use of 3.10 in section 3 but with the context needed for all three rules, thus the stack can contain both addresses and arguments. We will not show this intermediate machine but instead go straight to the result of also folding (SubL) into the system similarly to the use of 3.12. We only have to ascertain that the address inside each substitution is never removed by our strategy, then it is easy to see that all the rules fold without loss of sharing. This gives the machine in figure 4 for which we have the following:

4.10 Theorem. *For closed* M, $M \Downarrow_\ell \lambda x.R$ *iff* $(M, \epsilon, \epsilon) \xrightarrow[\ell\text{axgc}]{}\!\!\!\twoheadrightarrow (N, \epsilon, \epsilon)$ *and* $\downarrow_x \wr N \int \equiv \lambda x.R$.

Proof. Follows by a transformation correctness chain similar to that of section 3. □

Notice how the machine actually does the updating of the shared instances explicitly in (aSE). This corresponds to the update that a 'real' machine would *already* have done at the time where the reduction happened, however, (aSE) highlights that it is safe to *delay resyncronisation of the sharing* until a normal form is reached. Similarly, (xaSE) highlights the fact that substituting into a term copies it (the a is removed). This is the essential advantage of (xgcSE): it can remove the substitution without copying the subterm as a side effect – hence we can use this to reason about λ-lifting. In short insights about acyclic λ-graph reduction can be obtained from this system even though it was developed by theoretical means.

$$
\begin{array}{rll}
(\ M\langle x := N\rangle\ ,\ E\ ,\ S\ ) \to (\ M\ ,\ \langle x := N\rangle E\ ,\ S\ ) & & \text{(E)} \\
(\ MN\ ,\ E\ ,\ S\ ) \to (\ M\ ,\ E\ ,\ NE \cdot S\ ) & & \text{(SE)} \\
(\ M^a\ ,\ \epsilon\ ,\ S\ ) \to (\ M\ ,\ \epsilon\ ,\ a \cdot S\ ) & & \text{(SaE)} \\
(\ \lambda x.M\ ,\ \epsilon\ ,\ a \cdot S\ ) \to (\ \lambda x.M\ ,\ \epsilon\ ,\ S[(\lambda x.M)^a]\ ) & & \text{(aSE)} \\
(\ \lambda x.M\ ,\ \epsilon\ ,\ N \cdot S\ ) \to (\ M\ ,\ \langle x := N^a\rangle\ ,\ S\ ) & a \text{ fresh} & \text{(baSE)} \\
(\ x\ ,\ \langle x := N\rangle E\ ,\ S\ ) \to (\ N\ ,\ E\ ,\ S\ ) & & \text{(xvSE)} \\
(\ x\ ,\ \langle y := N\rangle E\ ,\ S\ ) \to (\ x\ ,\ E\ ,\ S\ ) & \text{if } x \not\equiv y & \text{(xvgcSE)} \\
(\ \lambda x.M\ ,\ \langle y := N\rangle E\ ,\ S\ ) \to (\ \lambda x.M\langle y := N\rangle E\ ,\ \epsilon\ ,\ S\ ) & & \text{(xabSE)} \\
(\ M^a\ ,\ \langle y := N\rangle E\ ,\ S\ ) \to (\ M\ ,\ \langle y := N\rangle E\ ,\ S\ ) & & \text{(xaSE)} \\
(\ M\ ,\ \langle y := N\rangle E\ ,\ S\ ) \to (\ M\ ,\ E\ ,\ S\ ) & \text{if } y \notin \mathrm{fv}(M) & \text{(xgcSE)}
\end{array}
$$

Figure 4: Abstract $\lambda\ell$-machine with sharing: $\lambda\ell$axgcSE.

5 Conclusions

We have shown how to model (acyclic) sharing. Current work includes the following:

- Using the notion of *explicit substitutes* of Rose (1995) we can generalise the sharing to combinatory reduction systems (CRS, Klop et al. 1993). Among other things this allows modelling of *cyclic substitutions* of Rose (1993a); in fact the resulting derivation gives an explanation of the notion of a 'black hole' (*aka* 'spine loop') from the possiblility in CRS of metaapplying a metavariable to itself (as in the CRS rule $\mu x.Z(x) \to Z(\mu x.Z(x))$).
- Comparing to λ-graph rewriting (Ariola and Klop 1994, Rose 1993b) where there is no distinction between variables and addresses, and to the 'call-by-need' calculus of Ariola, Felleisen, Maraist, Odersky and Wadler (1995): both seem to be expressible as special cases of the theorems (generalised to CRS).
- It is possible to generalise the labelling to allow proper treatment of the 'freshness' and (gc) side conditions.
- The relation between (gc), λ-lifting of Johnsson (1984), and 'trimming' of Sestoft (1994) should be investigated.

Future work includes generating 'machine code' corresponding to the steps of the abstract machine (for CRSs, in fact).

Acknowledgements. Thanks to Roel Bloo for collaboration on λxgc, and to Neil Jones and Eva Rose for their comments to the draft of this paper.

References

Abadi, M., Cardelli, L., Curien, P.-L. and Lévy, J.-J. (1991). Explicit substitutions. *Journal of Functional Programming* 1(4): 375–416.

Abramsky, S. (1990). The lazy lambda calculus. *In* D. A. Turner (ed.), *Research Topics in Functional Programming*. Addison-Wesley. chapter 4: pp. 65–116.

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M. and Wadler, P. (1995). A call-by-need lambda calculus. *POPL '95—22nd Annual ACM Symposium on Principles of Programming Languages*. San Francisco, California. pp. 233–246.

Ariola, Z. M. and Klop, J. W. (1994). Cyclic lambda graph rewriting. *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press. Paris, France. pp. 416–425.

Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics*. Revised edn. North-Holland.

Bloo, R. and Rose, K. H. (1995). Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. *CSN '95 – Computer Science in the Netherlands*. World Wide Web URL: *ftp://ftp.diku.dk/diku/semantics/papers/D-246.ps*

Curien, P.-L. (1986). Categorical combinators. *Information and Control* 69: 188–254.

Curien, P.-L. (1991). An abstract framework for environment machines. *Theoretical Computer Science* 82: 389–402.

de Bruijn, N. G. (1972). Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Koninklijke Nederlandse Akademie van Wetenschappen, Series A, Mathematical Sciences* 75: 381–392.

Hannan, J. and Miller, D. (1990). From operational semantics to abstract machines: Preliminary results. *1990 ACM Conference on LISP and Functional Programming*. Nice, France. pp. 323–332.

Hughes, J. M. (1982). Super-combinators: A new implementation method for applicative languages. *1982 ACM Symposium on LISP and Functional Programming*. Pittsburgh, Pensylvania. pp. 1–10.

Johnsson, T. (1984). Efficient compilation of lazy evaluation. *SIGPLAN Notices* 19(6): 58–69.

Kamareddine, F. and Ríos, A. (1995). A λ-calculus à la de Bruijn with explicit substitutions. *In* M. Hermenegildo and S. D. Swierstra (eds), *PLILP '95—Programming Languages: Implementation, Logics and Programs*. Number 982 in *LNCS*. Springer-Verlag. Utrecht, The Netherlands. pp. 45–62.

Klop, J. W. (1992). Term rewriting systems. *In* S. Abramsky, D. M. Gabbay and T. S. E. Maibaum (eds), *Handbook of Logic in Computer Science*. Vol. 2. Oxford University Press. pp. 1–116.

Klop, J. W., van Oostrom, V. and van Raamsdonk, F. (1993). Combinatory reduction systems: Introduction and survey. *Theoretical Computer Science* 121: 279–308.

Landin, P. J. (1964). The mechanical evaluation of expressions. *Computer Journal* 6: 308–320.

Launchbury, J. (1993). A natural semantics for lazy evaluation. *POPL '93—Twentieth Annual ACM Symposium on Principles of Programming Languages*. Charleston, South Carolina. pp. 144–154.

Lescanne, P. (1994). From λσ to λυ: a journey through calculi of explicit substitutions. *POPL '94—21st Annual ACM Symposium on Principles of Programming Languages*. Portland, Oregon. pp. 60–69.

Lévy, J.-J. (1978). *Réductions Correctes et Optimales dans le Lambda-Calcul*. Thèse d'etat. Université Paris 7.

Plotkin, G. D. (1981). A structural approach to operational semantics. *Technical Report FN-19*. DAIMI, Aarhus University. Aarhus, Denmark.

Rose, K. H. (1993a). Explicit cyclic substitutions. *Semantics Note D-166*. DIKU (University of Copenhagen). Universitetsparken 1, DK-2100 København Ø, Denmark. World Wide Web URL: *ftp://ftp.diku.dk/diku/semantics/papers/D-166.ps*

Rose, K. H. (1993b). Graph-based operational semantics of a lazy functional language. *In* M. R. Sleep, M. J. Plasmeijer and M. C. D. J. van Eekelen (eds), *Term Graph Rewriting: Theory and Practice*. John Wiley & Sons. chapter 22: pp. 303–316. World Wide Web URL: *ftp://ftp.diku.dk/diku/semantics/papers/D-146.ps*

Rose, K. H. (1995). Combinatory reduction systems with explicit substitution. *HOA '95 – Second International Workshop on Higher-Order Algebra, Logic and Term Rewriting*. Paderborn, Germany. World Wide Web URL: *ftp://ftp.diku.dk/diku/semantics/papers/D-247.ps*

Sestoft, P. (1994). Deriving a lazy abstract machine. *Technical Report ID-TR 1994-146*. Dept. of Computer Science, Technical University of Denmark.

Wadsworth, C. P. (1971). *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis. Programming Research Group, Oxford University.