BRICS

Basic Research in Computer Science

Modeling Sharing and Recursion for Weak Reduction Strategies using Explicit Substitution

Zine-El-Abidine Benaissa
Pierre Lescanne
Kristoffer H. Rose

BRICS Report Series **RS-96-56**

ISSN 0909-0878 **December 1996**

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through World Wide Web and anonymous FTP:

http://www.brics.dk/
ftp://ftp.brics.dk/
This document in subdirectory RS/96/56/

Modeling Sharing and Recursion for Weak Reduction Strategies using Explicit Substitution

Zine-El-Abidine Benaissa, Pierre Lescanne
INRIA Lorraine & CRIN, Nancy*

Kristoffer H. Rose
BRICS, University of Aarhus†

December 1996

*INRIA-Lorraine & CRIN, Bâtiment LORIA, 615, rue du Jardin Botanique, BP 101, F–54602 Villers les Nancy Cedex (France). E-mail: {benaissa,lescanne}@loria.fr.

†Basic Research in Computer Science (Centre of the Danish National Research Foundation), Dept. of Computer Science, University of Aarhus, Ny Munkegade bld.540, DK–8000 Aarhus C (Denmark). E-mail: krisrose@brics.dk.

Abstract

We *present* the $\lambda\sigma_w^a$-calculus, a formal synthesis of the concepts of sharing and explicit substitution for weak reduction. We show how $\lambda\sigma_w^a$ can be used as a foundation of implementations of functional programming languages by modeling the essential ingredients of such implementations, namely *weak reduction strategies*, *recursion*, *space leaks*, *recursive data structures*, and *parallel evaluation*, in a uniform way.

First, we give a precise account of the major reduction strategies used in functional programming and the consequences of choosing λ-graph-reduction vs. environment-based evaluation. Second, we show how to add *constructors and explicit recursion* to give a precise account of recursive functions and data structures even with respect to space complexity. Third, we formalize the notion of *space leaks* in $\lambda\sigma_w^a$ and use this to define a space leak free calculus; this suggests optimisations for call-by-need reduction that prevent space leaking and enables us to prove that the "trimming" performed by the STG machine does not leak space.

In summary we give a formal account of several implementation techniques used by state of the art implementations of functional programming languages.

Keywords. Implementation of functional programming, lambda calculus, weak reduction, explicit substitution, sharing, recursion, space leaks.

Contents

1 Introduction

The aim of this paper is to present a framework for several forms of implementation of functional programming languages. It is often said that there are essentially two main classes of implementation, namely those based on *graph reduction* and those based on *environments*. The first ones are efficient in that they optimize the code sharing, the second in that they allow a design of the implementation closer to the hardware. These two classes are traditionally split into subclasses according to the strategy which is used (evaluation of the value first, of the functional body first, or by need). However, in our approach a strategy is not an ingredient of a specific implementation, but something which is defined fully independently of the chosen form of implementation (graph reduction or environments). Our unifying framework is a calculus which describes faithfully and in detail all the mechanisms involved in weak reduction. Rather naturally we have chosen a weak calculus of explicit substitution (Curien, Hardin & Lévy 1992) as the basis, extending it with *global addresses*. This way, we can talk easily, and at a high level of detail, about addresses and sharing. At first, the calculus is not tied to any kind of implementation: the separation between graph reduction and environment-based evaluation emerges naturally when studying how to assign an address to the result of a successful variable lookup. Strategies come later as restrictions of the calculus. In this paper we study *call-by-value* and *call-by-name* of Plotkin (1975) as well as the not yet fully understood *call-by-need* strategy, and we show how parallel strategies can be devised. Finally we treat an important problem of implementation for which people propose ad-hoc solutions, namely *space leaking*: we propose a natural and efficient solution to prevent it (which we prove correct).

1.1 Complexity of functional computations

Computer science is rooted in the notion of *computability* and *decidability* which was developed in the 30s by Turing (1936) and Church (1936) as an answer to Hilbert and Gödel (1931). While the purpose of these studies was merely to define the "effectively computable" functions by establishing what could *possibly* be computed in a mechanical way, the formulation still ended up having a profound influence on what was to be known as "high-level programming languages" since few other notations were (and are) known for expressing problems in a way that is guaranteed to be computable.

For the languages inspired by "Turing Machines" (TMs), now known as *imperative* programming languages, there is a natural notion of a computation step. As a consequence, the study of computational *complexity* of algorithms expressed as imperative programs is well established, and precise measures for the time (and space) resources required to solve many computable problems exist, all based on the assumption that the step (and data structure) of a TM provides a fair unit of computational time (and space).

For languages based on Church's "λ-calculus" the issue is not so clear, however: there never was a properly established notion of "computation step" for the λ-calculus: the original notion of contraction using the rule

$$(\lambda x.M)N \to M[x := N] \tag{β}$$

was recognized from the start as not fulfilling this purpose, because it is not clear that there is a bound on the amount of work involved in the substitution of N for x in M since M can contain an arbitrary number of xs. This was recognized as a problem and a first solution was found: *combinatory logic* introduced by Schönfinkel (1924) and (see Curry & Feys (1958) for a discussion Curry 1930) gives a truly stepwise realization of the λ-calculus and hence assigns a complexity to reduction of λ-terms. However, this measure is far too pessimistic for realistic use for two reasons: first it *duplicates* computation, second it strictly enforces a particular *sequence* of evaluation, third the coding of λ-terms into combinatory logic *changes the structure* of the term in a way that may increase the size quadratically. Thus for a while there was no real way to assign complexity to algorithms based on their formulation in λ-calculus (or languages based on λ-calculus – the *functional languages*[1]).

1.2 Sharing and strategies

The seminal fundamental study of the complexity of λ-evaluation was Landin's (1965) *SECD machine* which was the first device described to mechanically evaluate λ-terms. The technique used is as follows. First one translates the λ-expressions into a sequence of "instructions", second one reduces those instructions sequentially using a complicated system of stacks to keep track of the code to be executed later. The complexity boils down to a measure w.r.t.

[1]Strictly speaking, functional languages are based on the notion of *recursive equations* as explained by McCarthy (1960) which is λ-calculus plus an explicit recursion operator; we return to this subtle point in the paper.

a specific *strategy* and makes the mentioned *duplication* implicit in the data structures. Later abstract machines include some variation in the used reduction strategy but remain dedicated to one (some early, seminal examples are Plotkin 1977, Henderson 1980, Cardelli 1983).

The first issue, *sharing* to avoid duplication, was addressed already by Wadsworth (1971) who proposed λ-*graph reduction* which is the simple idea that duplication should be *delayed* as long as possible by representing subterms of common origin by identical subgraphs. This way all "duplicates" profit from any reductions happening to that particular subterm (or -graph, as it were). This technique was combined with combinator reduction by Turner (1979) and generalized by Hughes (1982) into *supercombinator graph reduction* which is the computational model underlying most implementations of functional languages today.

The second issue, evaluation sequence, was addressed partly by Plotkin (1975) who identified the difference between the evaluation sequence or *reduction strategy* used in all implementations at the time, *call-by-value*, and the simplest normalizing strategy known to normalize, *call-by-name*. The issue of the complexity of evaluation was not addressed, however. The work on *categorical combinatory logic* introduced by Curien (1983) and the following *explicit substitutions* (Abadi, Cardelli, Curien & Lévy 1991) permitted this to be repaired: here any strategy can be used and the complexity in each case is represented by the *reduction length* in that substitution is defined in a *stepwise* manner such that each step can conceivably be implemented in *constant time* and hence serve as the basis for the study and systematic derivation of efficient implementations.

1.3 Generic descriptions of implementations

The aim of providing an accurate analysis of the complexity of computations of functional programs including achieving a clear and independent integration of sharing and description of the restriction to a strategy, has lead us to a new approach to the description of abstract machines. Indeed, if those three aspects can be fully dissociated we can to propose a generic description of abstract machines, yielding each specific machine as a particular instantiation of several parameters. This is clearly illustrated by Table 4. In this convenient framework we were able to formalize other aspects of functional programming languages, namely recursive definitions, constructors and space leak freeness.

1.4 Plan

We achieve our aim by providing a solution to the three issues exemplified in the previous sections: we obtain a computational models that, in a general fashion, is a realistic computational model for weak λ-calculus reduction incorporating *both* realistic sharing to avoid duplicating work *and* a realistic measure of the complexity of substitution and leading to a generic description of abstract machines

We start in Section 3 by combining sharing and explicit substitution for weak λ-calculus (reflecting that functional languages share the restriction that reduction never happens under a λ) into a calculus, $\lambda\sigma_w^a$, with explicit substitution, naming, and addresses. Moreover, it naturally permits two update principles that are readily identifiable as *graph reduction* (Wadsworth 1971) and *environment-based evaluation* (Curien 1991). In Section 4 we show how $\lambda\sigma_w^a$ adequately describes sharing with any (weak) reduction strategy; the proof is particularly simple because it can be tied directly to addresses; to illustrate this we prove that $\lambda\sigma_w^a$ includes the "λ_{let}" calculus of Ariola, Felleisen, Maraist, Odersky & Wadler (1995). In Section 5 we study how the usual extensions of *explicit recursion* and *data constructors* can be added to give the full expressive power of functional programming. In Section 6, we illustrate the adaptability of this calculus by defining the notion of a *space leaking*, and we study a class of *space leak free* subcalculi. As a corollary we get that the trimming used in the "STG" calculus of Peyton Jones (1992) is safe.

2 Preliminaries

We start by summarizing certain standard concepts and notations that will prove convenient throughout.

Notation 2.1 (relations). We designate *binary relations* by arrows, following Klop (1992). Let $\rightarrow, \underset{1}{\rightarrow}, \underset{2}{\rightarrow} \subseteq \mathbf{A} \times \mathbf{A}$ be such relations and let $a, b \in \mathbf{A}$.

1. $\underset{1+2}{\longrightarrow} = \underset{1}{\rightarrow} \cup \underset{2}{\rightarrow}$ and $\underset{1}{\rightarrow} \cdot \underset{2}{\rightarrow}$ denotes the composition of $\underset{1}{\rightarrow}$ and $\underset{2}{\rightarrow}$.

2. $\twoheadrightarrow$ is the *transitive reflexive closure* of $\rightarrow$.

3. We use Rosen's (1973) *stencil diagrams* to express propositions (several examples are given below), with solid arrows and nodes denoting

(outer) universally quantified relations and objects, and dotted arrows and hollow nodes denoting (inner) existentially quantified relations and objects, respectively.

4. $\rightarrow$ is *confluent* if $\twoheadleftarrow \cdot \twoheadrightarrow \subseteq \twoheadrightarrow \cdot \twoheadleftarrow$.

5. The $\rightarrow$-*normal forms* are those a such that $\nexists b : a \rightarrow b$. In other words, a is a normal form if $a \twoheadrightarrow b$ implies $a = b$.

6. $\twoheadrightarrow\!\!\!|$ is the *normal form restriction* of $\twoheadrightarrow$ that satisfies $a \twoheadrightarrow\!\!\!| \; b$ iff $a \twoheadrightarrow b$ and b is a $\rightarrow$-normal form.

7. $\rightarrow$ is *terminating* if all sequences $a_1 \rightarrow a_2 \rightarrow \cdots$ are finite; it is *convergent* if it is terminating and confluent.

Definition 2.2 (term rewriting). The following summarizes the main concepts of term rewriting used in this paper.

1. We permit inductive definition of sets of *terms* using syntax productions. Furthermore, we write $C\{\_\}$ for a *context*.

2. A *term rewrite system* is a set rules $\ell_i \rightarrow r_i$, where ℓ_i and r_i are terms called the *left-hand side* (lhs) and r_i the *right-hand side* (rhs), respectively. Furthermore, all variables in a rhs also occur in the associated lhs.

3. A *substitution* σ is a map from variables to terms; this is homeomorphically extended to any term t such that $\sigma(t)$ denotes the term obtained by replacing all variables x in t with $\sigma(x)$. A *redex* is a term of the form $\sigma(\ell)$ for some substitution σ and some rewrite rule $\ell \rightarrow r$. A *rewrite step* is the procedure of replacing a redex in some context $C\{\_\}$ with its *reduct* $C\{\sigma(r)\}$; we then write $C\{\sigma(\ell)\} \rightarrow C\{\sigma(r)\}$.

4. We say that two rewrite rules $\ell_1 \rightarrow r_1$ and $\ell_2 \rightarrow r_2$ *overlap* if there exists substitutions σ_1, σ_2, a (possibly trivial) context $C_1\{\_\}$, and a term t_2, such that $\sigma_1(\ell_1) = C_1\{t_2\}$ and $\sigma_2(\ell_2) = t_2$ but $\nexists\sigma\colon \sigma(\ell_1) = C_1\{x\}$. A term rewrite system is *orthogonal* if it has no overlaps and is *left-linear* (no variable occurs more than once in any lhs).

5. Let the overlining of a term t, $\overline{t}$, be obtained by replacing all the symbols in t with (new and unique) "overlined" symbols. For a term rewrite

system R with rewrite rules $\ell \to r$ let $\overline{R}$ have rules $\overline{\ell} \to r$. Clearly, if we overline the symbols of a redex for the R-rule $\ell \to r$ in a term t, then this will now instead be a redex for the $\overline{R}$-rule $\overline{\ell} \to r$. If t contains several $\overline{R}$-rule redexes, in particular one such that $t = C\{\sigma(\overline{\ell})\}$ for the $\overline{R}$-rule $\overline{\ell} \to r$, then the $\overline{R}$-redexes remaining in the reduct $C\{\sigma(r)\}$ are called the *residuals* of the non-reduced $\overline{R}$-redexes. Reducing to $\overline{R}$-normal form and then erasing all overlines is called a *development*; developing $\overline{t}$ is called a *complete development* since all redexes of t as well as all the residuals are reduced.

Notation 2.3 (de Bruijn notation). We employ the "namefree" representation of λ-terms invented by de Bruijn (1972): Each occurrence of a variable in a λ-term is represented by a natural number, called the *index*, corresponding to the number of λ's traversed from its binding λ to it (so indices start at 0). The set of these terms is defined inductively by $M, N ::= \lambda M \mid MN \mid \underline{n}$. The *free indices* of a term correspond to the indices at the outermost level of what is usually known as the *free variables* and is given by $\mathrm{fi}(M) = \mathrm{fi}_0(M)$ where fi_i is again defined inductively by $\mathrm{fi}_i(MN) = \mathrm{fi}_i(M) \cup \mathrm{fi}_i(N)$, $\mathrm{fi}_i(\lambda M) = \mathrm{fi}_{i+1}(M)$, and $\mathrm{fi}_i(\underline{n}) = \{n - i\}$ when $n \geq i$ but $\varnothing$ when $n < i$. We call this calculus λ_{NF} and when mixing it with other calculi we will refer to λ_{NF}-terms as *pure* terms. In what follows, a *value* is a term of the form $\lambda M[s]$ or $\underline{n}\, V_1 \ldots V_m$ where $V_1, \ldots, V_m$ are also values.

Finally, we base our work on the $\lambda\sigma_w$-calculus, shown in Fig. 1, one of several calculi of *weak explicit substitution* given by Curien et al. (1992). The idea behind this calculus is to forbid substitution in abstractions: this is accomplished by never propagating explicit substitutions inside abstractions yet requiring in (B_w) that a substitution is present in every redex. This restriction gives a confluent weak calculus but necessitates using a term representation with lists of bindings as originally found in the $\lambda\rho$-calculus of Curien (1991).

3 Calculi for Weak Reduction with Sharing

In this section we generalize the weak explicit substitution calculus $\lambda\sigma_w$ defined by Curien et al. (1992) to include addresses in order to explicit pointer manipulations. Our starting point is reminiscent of the labeling used by

Terms. M and N denote pure λ_{NF}-terms. $\lambda\sigma_w$-terms are ranged over by W given by

$$W ::= \underline{n} \;\big|\; WW \;\big|\; M[s] \qquad \text{(Weak)}$$

$$M, N ::= \lambda M \;\big|\; MN \;\big|\; \underline{n} \qquad \text{(Pure)}$$

$$s ::= W \cdot s \;\big|\; \mathrm{id} \qquad \text{(Substitution)}$$

Beta-reduction.

$$(\lambda M)[s]\, W \to M[W \cdot s] \qquad (\mathrm{B}_w)$$

Substitution elimination.

$$(MN)[s] \to M[s]\, N[s] \qquad \text{(App)}$$

$$\underline{0}\,[W \cdot s] \to W \qquad \text{(FVar)}$$

$$\underline{n+1}\,[W \cdot s] \to \underline{n}\,[s] \qquad \text{(RVar)}$$

$$\underline{n}\,[\mathrm{id}] \to \underline{n} \qquad \text{(VarId)}$$

Figure 1: $\lambda\sigma_w$.

Maranget (1991),[2] however, our notion of "address" is more abstract and allows us a better comprehension of implementations of machines for functional programming languages and their optimizations.

3.1 Explicit substitution and sharing

Figure 3 presents the syntax and reduction rules of $\lambda\sigma_w^a$ (it will be properly stated in Definition 3.10 once the notion of address is formally established). Like $\lambda\sigma_w$ it forbids substitution in abstractions by never propagating explicit substitutions inside abstractions yet requiring that every redex must contain a substitution. This restriction gives a confluent weak calculus but necessitates using a term representation with lists of bindings as originally found in the $\lambda\rho$ calculus of Curien (1991).

$\lambda\sigma_w^a$ includes the special rule (Collect) which is meant to save useless computations and can be omitted: it collects "garbage" in the style of Bloo & Rose (1995), *i.e.*, terms in substitutions which are never substituted. Although computation in such useless terms can be avoided using specific strategies, the accumulation of useless terms in substitutions is a drawback w.r.t. the size of terms, and hence w.r.t. space complexity. In Section 6, we study a phenomenon well known in functional programming as the *space leak problem*.

Furthermore, notice that the rules (FVarG) and (FVarE) have the same left-hand side (LHS): the difference between these two rules is in the choice of the address in the right-hand side (RHS) which is either b (FVarG) or a (FVarE). This conceptual choice has a direct correspondence to the duplication versus sharing problem of implementations. This is illustrated by the example in Fig. 2. We obtain four possible different resulting terms namely $(\underline{n}^b\,\underline{n}^b)^a$, $(\underline{n}^b\,\underline{n}^d)^a$, $(\underline{n}^c\,\underline{n}^b)^a$, and $(\underline{n}^c\,\underline{n}^d)^a$. It is clear that erasing the addresses of these four terms produces the same term, namely $\underline{n}\,\underline{n}$: the difference between them is the amount of sharing (or shapes of the associated graph) we obtain, more precisely, *the use of* (FVarG) *maintains sharing whenever* (FVarE) *decreases it.* Also notice that sharing is only increased when a term with addresses in it is duplicated, *i.e.*, when (App) is used with some addresses inside s. As a consequence, further (parallel) rewriting of this argument will be shared with all its other occurrences in the term: Assume that the address is a, a copy of the term E at address b (or, to be precise,

[2] In particular the use of developments and parallel reduction to model sharing.

$$
\begin{tikzcd}[column sep=large]
 & & (\underline{n}^b\,\underline{n}^b)^a \\
 & (\underline{n}^b\,\underline{0}\,[\underline{n}^b]^d)^a \arrow[ur, "\text{(FVarG)}"] \arrow[r, "\text{(FVarE)}"] & (\underline{n}^b\,\underline{n}^d)^a \\
(\underline{0}\,[\underline{n}^b]^c\;\underline{0}\,[\underline{n}^b]^d)^a \arrow[ur, "\text{(FVarG)}"] \arrow[dr, "\text{(FVarE)}"'] & & \\
 & (\underline{n}^c\,\underline{0}\,[\underline{n}^b]^d)^a \arrow[r, "\text{(FVarG)}"] \arrow[dr, "\text{(FVarE)}"'] & (\underline{n}^c\,\underline{n}^b)^a \\
 & & (\underline{n}^c\,\underline{n}^d)^a
\end{tikzcd}
$$

Figure 2: Graph- *vs.* Environment reduction.

a copy of its root node because addresses of its subterms are not changed) is performed. Then the term $\underline{0}\,[E \cdot s]$ at address a is replaced by that copy and later rewriting of this argument will not be shared. Thus we see that the reduction $\underset{\sigma}{\rightarrow}$ contains two substitution elimination subreductions, $\underset{\sigma g}{\longrightarrow}$ and $\underset{\sigma e}{\longrightarrow}$. Let $\sigma_0 = \{(\mathrm{App}), (\mathrm{RVar}), (\mathrm{VarId})\}$. Then $\sigma g = \sigma_0 \cup \{(\mathrm{FVarG})\}$ and $\sigma e = \sigma_0 \cup \{(\mathrm{FVarE})\}$.

3.2 Addresses and parallel reduction

So, a consequence of mixing those two systems is the creation of a critical pair (non-determinism) and thus non-orthogonality. Fortunately, since this critical pair is at the root, the residual redex notion (Huet & Lévy 1991) can be extended in a straight-forward way: We just observe that there is no residual redex of (FVarG) (resp. (FVarE)) after applying (FVarE) (resp. (FVarG)). We first establish that this is safe before we give the definition (Def. 3.7).

Definition 3.1. A *complete development* of a preterm T is a series of $\lambda\sigma^a_w$-rewrites that rewrite all redexes of T until no residuals remain.

Thus the non deterministic choice between (FVarE) and (FVarG), discussed above, makes complete development *nondeterministic*. We will denote the set of preterms obtained by all possible complete developments of T by $\mathrm{dev}(T)$; notice that these preterms depend on the fresh addresses introduced by (App).

Lemma 3.2. $\mathrm{dev}(T)$ *is finite for any preterm* T*.*

Syntax. The *addressed preterms* are defined inductively by

$$
\begin{aligned}
T, U, V &::= E^a \mid \bot && \text{(Addressed)}\\
E, F &::= M[s] \mid UV \mid \underline{n} && \text{(Evaluation Context)}\\
M, N &::= \lambda M \mid MN \mid \underline{n} && \text{(Pure)}\\
s, t &::= \mathrm{id} \mid U \cdot s && \text{(Substitution)}
\end{aligned}
$$

where everywhere a, b, c range over an infinite set $\mathbf{A}$ of *addresses*.

Weak β-introduction prereduction. $\xrightarrow[\mathrm{B}_w]{}$ is defined by the rule

$$((\lambda M)[s]^b\, U)^a \to M[U \cdot s]^a \qquad (\mathrm{B}_w)$$

Weak substitution elimination prereduction. $\xrightarrow[\sigma]{}$ is defined by the rules

$$
\begin{aligned}
(MN)[s]^a &\to (M[s]^b\, N[s]^c)^a \qquad b, c \text{ fresh} && \text{(App)}\\
\underline{0}\,[E^b \cdot s]^a &\to E^b && \text{(FVarG)}\\
\underline{0}\,[E^b \cdot s]^a &\to E^a && \text{(FVarE)}\\
\underline{n+1}\,[U \cdot s]^a &\to \underline{n}\,[s]^a && \text{(RVar)}\\
\underline{n}\,[\mathrm{id}]^a &\to \underline{n}^a && \text{(VarId)}
\end{aligned}
$$

Collection prereduction. $\xrightarrow[\mathrm{C}]{}$ is defined by the rule

$$M[s]^a \to M[s|_{\mathrm{fi}(M)}]^a \qquad s \neq s|_{\mathrm{fi}(M)} \qquad \text{(Collect)}$$

where *environment trimming* is defined by $s|_I = s|_I^0$ where $\mathrm{id}|_I^i = \mathrm{id}$ and $(U \cdot s)|_I^i = U \cdot s|_I^{i+1}$ when $i \in I$ but $\bot \cdot s|_I^{i+1}$ when $i \notin I$.

Figure 3: $\lambda\sigma_w^a$: syntax and reduction rules.

Proof. Clearly $\#\mathrm{dev}(T) \leq 2^i$ where i is the number of (FVarG/E)-redexes in T. $\square$

Definition 3.3 (sharing ordering). Let θ be a map on the set of addresses. The ordering $\trianglerighteq_\theta$ is defined inductively by

- If for all i, $0 \leq i \leq n$ $T_i \trianglerighteq_\theta T'_i$ then $M[T_0 \cdots T_n]^a \trianglerighteq_\theta M[T'_0 \cdots T'_n]^{\theta(a)}$,
- $n^a \trianglerighteq_\theta n^{\theta(a)}$, and
- if $T \trianglerighteq_\theta T'$ and $U \trianglerighteq_\theta U'$ then $(TU)^a \trianglerighteq_\theta (T'U')^{\theta(a)}$.

we say that the addressed term T collapses in U, $T \trianglerighteq U$ if there exists a map θ such that $U \trianglerighteq_\theta (T)$.

Lemma 3.4. *Let T be a preterm.* $(\mathrm{dev}(T), \trianglerighteq)$ *is a finite complete partial ordering (cpo).*

Proof. The lower bound of two preterms T_1 and T_2 belonging to $\mathrm{dev}(T)$ is the most general unifier of T_1 and T_2 where the addresses are interpreted as free variables (and the result of course considered modulo renaming). Moreover, $\trianglerighteq$ is a partial ordering. $\square$

The next lemma uses the translation function "erase" which deletes all addresses of an addressed term, obviously resulting in a $\lambda\sigma_w$-term.

Lemma 3.5. *Let P and Q be two preterms. If $P \xrightarrow[\lambda\sigma_w^a]{} Q$ then* $\mathrm{erase}(P) \xrightarrow[\lambda\sigma_w]{}$ $\mathrm{erase}(Q)$*.*

Proof. Obvious. $\square$

Theorem 3.6 (finite development). *Let T be a preterm. Then all complete developments of T are finite*

Proof. From Lemma 3.5, and the fact that $\lambda\sigma_w$ is orthogonal, we obtain that $\lambda\sigma_w^a$ has the finite development property. $\square$

Now we can define the parallel extension $\underset{R}{\twoheadrightarrow}$ of a rewrite system R which corresponds intuitively to the simultaneous reduction of all R-redexes of a term T, *e.g.*, one $\underset{\mathrm{B}_w}{\twoheadrightarrow}$ step is the simultaneous reduction of all B_w-redexes.

Definition 3.7 (parallel extension). Let T and U be two terms. Then the parallel extension $\mapsto\!\!\!\rightarrow$ of the rewrite relation $\rightarrow$ is defined by $T \mapsto\!\!\!\rightarrow U$ iff $U \in \mathrm{dev}(T)$.

The next definition shows how to determine when an 'addressed term' corresponds to a 'directed acyclic graph:' this should be the case exactly when the 'sharing information' is non-ambiguous.

Definition 3.8 (addressing). Given some set of terms $\mathbf{T}$ with a notion of *address* associated to some subterms (written as superscripts). Assume $t \in \mathbf{T}$ is such a term and let $a, b \in \mathbf{A}$ range over the possible addresses.

1. The set of all addresses that occur in t is written $\mathrm{addr}(t)$.

2. The *outermost a-addressed subterms* of t is the set $t@a = \{s_1^a, \ldots, s_n^a\}$ for which an n-ary context $C\{\,\}$ exists such that $t = C\{s_1^a, \ldots, s_n^a\}$ and $a \notin \mathrm{addr}(C\{\_, \ldots, \_\})$.

3. t is *admissible* if all addresses $a \in \mathrm{addr}(t)$ satisfy $t@a = \{s^a\}$ where $a \notin \mathrm{addr}(s)$ (this is a variant of a notion of Wadsworth 1971).

4. If $\rightarrow$ is a rewrite relation on $\mathbf{T}$ defined by a certain number of axioms, then for each address a, $\xrightarrow{a}$ is the *address restriction* of $\rightarrow$ to only those (proper) reductions where the redex has address a. This is generalized to any set of addresses $A = \{a_1, \ldots, a_n\}$: $\xrightarrow{A} = \bigcup_{a \in A} \xrightarrow{a}$.

5. $\overset{a}{\mapsto\!\!\!\rightarrow}$ (resp. $\overset{A}{\longmapsto\!\!\!\rightarrow}$) is the parallel extension of $\xrightarrow{a}$ (resp. $\xrightarrow{A}$).

6. Parallel $\rightarrow$-reduction is $\overset{\infty}{\longmapsto\!\!\!\rightarrow} = \bigcup_{A \subseteq \mathbf{A}} \overset{A}{\longmapsto\!\!\!\rightarrow}$.

7. n-parallel $\rightarrow$-reduction is $\overset{n}{\longmapsto\!\!\!\rightarrow} = \bigcup_{A \subseteq \mathbf{A} \wedge \# A \leq n} \overset{A}{\longmapsto\!\!\!\rightarrow}$; in particular $\overset{1}{\longmapsto\!\!\!\rightarrow}$ is called *serial* reduction.

Parallel reduction $\overset{\infty}{\longmapsto\!\!\!\rightarrow}$ expresses not only sharing but also parallel computation because at each step a parallel computer can reduce a set of addresses simultaneously. Notice that if $a \notin addr(U)$ or $A \cap addr(U) = \varnothing$ the reductions degenerate: $\{\, T \mid U \xrightarrow{a} T \,\} = \{\, T \mid U \overset{a}{\mapsto\!\!\!\rightarrow} T \,\} = \{T \mid U \xrightarrow{A} T\} = \{T \mid U \overset{A}{\longmapsto\!\!\!\rightarrow} T\} = \varnothing$.

Proposition 3.9. $\xmapsto[\mathrm{B}_w+\sigma+\mathrm{C}]{\infty}$ *preserves admissibility.*

Proof. By definition of parallel rewriting, one rewrite step rewrites all occurrences of an address a, hence admissibility is preserved. $\square$

3.3 Explicit substitution with addresses

Definition 3.10 ($\lambda\sigma_w^a$). Given the preterms and prereductions of Fig. 3.

1. The $\lambda\sigma_w^a$-*terms* are the admissible $\lambda\sigma_w^a$-preterms.
2. $\lambda\sigma_w^a$-*substitution* is the relation $\xmapsto[\sigma]{\infty}$,
3. $\lambda\sigma_w^a$-*reduction* is the relation $\xmapsto[\mathrm{B}_w+\sigma+\mathrm{C}]{\infty}$ which we will write as $\xmapsto{\infty}$ when confusion is unlikely.

As for $\lambda\sigma_w$ one associates with the pure term M the $\lambda\sigma_w^a$-term $M[\mathrm{id}]^a$ which will then reduce to weak normal form modulo substitution under abstraction. Normal forms, values V, are of the form $\lambda M[V_1 \cdots V_n]^a$ or $(\ldots(\underline{n}^a V_1)^{b_1} \ldots V_m)^{b_m}$.

The last two lemmas of this section establish the connection between $\lambda\sigma_w^a$ and $\lambda\sigma_w$ and together show the *correctness* and *confluence modulo erasure* of $\lambda\sigma_w^a$.

Lemma 3.11 (projection). *Let T and U be two addressed terms. If $T \xmapsto{\infty} U$ then* $\mathrm{erase}(T) \xrightarrow[\lambda\sigma_w]{} \!\!\!\!\rightarrow \mathrm{erase}(U)$.

Proof. Easy induction on the structure of T.

- $T = M[s]^b$, two cases are possible. If $a = b$, by case analysis on $\lambda\sigma^a$-rule, three rules can be applied namely (App), (FVarE), (FVarG), or (RVar). If (App) is applied (that means that $M = M_1M_2$) then

  $$\mathrm{erase}(T) = M_1M_2[\mathrm{erase}(s)] \xrightarrow[\lambda\sigma]{} M_1[\mathrm{erase}(s)]M_2[\mathrm{erase}(s)] = \mathrm{erase}(U)$$

  if $a \neq b$, then all evaluation contexts labeled by a are subterms of T, and are equals and induction hypothesis applies.

- $T = (T_1T_2)^b$, if $a = b$ only B_w rule can be applied. if $a \neq b$ then the induction hypothesis applies. $\square$

Lemma 3.12. *let W and W' be two $\lambda\sigma$-terms. If $W \xrightarrow[\lambda\sigma_w]{} W'$ then there exists two addressed terms T and U such that $W = \mathrm{erase}(T)$, $W' = \mathrm{erase}(U)$, and $T \overset{\infty}{\mapsto} U$.*

Proof. Label each subterm of W by its access path to ensure that each subterm has a unique address and then rewrite using the redex's address. More formally, we define the function (we called label) that translates $\lambda\sigma$-terms to evaluation contexts. This function assigns to each weak subterm of a weak $\lambda\sigma$-term its position relative to the root ensuring the uniqueness of addresses to subterms.

$$\begin{aligned}
\mathrm{label}(T_1T_2)_p &= (\mathrm{label}(T_1)_{1p}\mathrm{label}(T_2)_{2p})^p \\
\mathrm{label}(M[T_1 \cdots T_n \cdot \mathrm{id}])_p &= M[\mathrm{label}(T_1)_{1p} \cdots \mathrm{label}(T_n)_{np} \cdot \mathrm{id}]^p \\
\mathrm{label}(\,\underline{n}\,_p) &= \underline{n}^p
\end{aligned}$$

Then it is easy to show that $\mathrm{label}(M) \overset{a}{\mapsto} \mathrm{label}(N)$. □

Notice that label is a naive translation function of $\lambda\sigma$-terms because it does not profit from the possible amount of sharing in $\lambda\sigma^a$. For instance, let R be a weak $\lambda\sigma$-term then $\mathrm{label}(RR) = R^a R^b$ and thus reductions in the left R and in the right will not be done simultaneously because they have different addresses.

4 Reduction Strategies

In this section we show how conventional weak reduction strategies can be described through restrictions of the $\lambda\sigma_w^a$-calculus. First we formalize the classic sequential strategies call-by-name, call-by-value, and call-by-need. Second, we formalize two common parallel reduction strategies: spine parallelism and speculative parallelism.

4.1 Address-controlled strategies

The crucial difference from traditional expositions in both cases is that we can exploit that *all redexes have a unique address.* We will thus define a reduction with a strategy as a two-stage process (which can be merged in actual implementations, of course): we first give an inference system for "locating" a set of addresses where reduction can happen, and then we reduce using the original reduction constrained to just those addresses.

Definition 4.1 (strategy for addressed terms). Given a reduction $\xrightarrow[X]{}$ on a set of addressed terms.

1. A *strategy* $\mathcal{S}$ is a relation written $U \underset{\mathcal{S}}{\vdash} A$ from addressed terms U to sets of addresses A.

2. For a strategy $\mathcal{S}$, $\mathcal{S}$*-reduction* $\xrightarrow[X/\mathcal{S}]{}$ is defined by $U_1 \xrightarrow[X/\mathcal{S}]{} U_2$ iff $U_1 \underset{\mathcal{S}}{\vdash} A$ and $U_1 \xrightarrow[X]{A} U_2$.

We begin with the possible *reduction in context* inference rules which make possible to restrict the addresses at which the rewrite rules can be applied.

Definition 4.2 (strategy rules). The $\lambda\sigma_w^a$*-strategy rules* are the following:

$$\frac{}{M[s]^a \vdash \{a\}}\ (\mathrm{Scl}) \qquad \frac{U \vdash A_1 \quad U \vdash A_2}{U \vdash A}\ A = A_1 \cup A_2 \quad (\mathrm{Par})$$

$$\frac{s \vdash A}{M[s]^a \vdash A}\ (\mathrm{Sub}) \qquad \frac{U \vdash A}{U \cdot s \vdash A}\ (\mathrm{Hd}) \qquad \frac{s \vdash A}{U \cdot s \vdash A}\ (\mathrm{Tl})$$

$$\frac{}{(UV)^a \vdash \{a\}}\ (\mathrm{Sap}) \qquad \frac{U \vdash A}{(UV)^a \vdash A}\ (\mathrm{Lap}) \qquad \frac{V \vdash A}{(UV)^a \vdash A}\ (\mathrm{Rap})$$

where we furthermore require for (Scl) and (Sap) that some $\lambda\sigma_w^a$-rule is, indeed, applicable. A $\lambda\sigma_w^a$-strategy $\mathcal{S}$ is specified by giving a list of conditions for application of each rule; this defines the relation $\underset{\mathcal{S}}{\vdash}$. Notice that the normal forms (or values) for some strategies are not $\lambda\sigma_w^a$-values, *i.e.*, closed weak normal forms. For instance, if (Rap) is disallowed then normal forms correspond to closed weak head normal forms (whnf) of shape $(\lambda M)[s]^a$.

Figure 4 shows the conditions for several strategies. Notice that when (Sub) is disabled then (Hd) and (Tl) are not reachable.

For the remainder of the section we will discuss these strategies. Notice that there are two forms of non determinism in the strategies. One is due to (Par), the only rule which contains the union operator and can yield more than one address for reduction. The other form of nondeterminism already

Strategy	(FVar?)	(Scl)	(Par)	(Sub)	(Hd)	(Tl)	(Sap)	(Lap)	(Rap)
$\checkmark$	E or G	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
CBN	E	$\checkmark$	$\times$	$\times$			①	$\neg$①	$\times$
CBV	E	$\checkmark$	$\times$	$\times$			① $\wedge$ ②	$\neg$①	① $\wedge\neg$②
CBNeedE	E	③	$\times$	$\neg$③	$\checkmark$	$\times$	①	$\neg$①	$\times$
CBNeedG	G	$\checkmark$	$\times$	$\times$			①	$\neg$①	$\times$
Specul-$\|_n$	E	$\checkmark$	④	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$
Spine-$\|_n$	E	$\checkmark$	④	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\times$

$\checkmark$: "always applicable;" $\times$: "never applicable;" blank: "don't care;" ①: U is a value; ②: V is a value; ③: if $s = U \cdot s'$ and $M = \underline{0}$ and U is a value; and ④: $\#A \leq n$.

Figure 4: Strategies for $\lambda\sigma_w^a$.

mentioned in Section 3 (and here permitted only in the first strategy) is the choice between (FVarE) and (FVarG).

The first strategy, "$\checkmark$," allows every rule and gives us the full $\lambda\sigma_w^a$-calculus.

Proposition 4.3. $\underset{X/\checkmark}{\longmapsto} = \underset{X}{\longmapsto}$.

Proof. An easy induction over $\lambda\sigma_w^a$-terms shows that $T \underset{\checkmark}{\vdash} A$ if and only if $\varnothing \subset A \subseteq \mathrm{addr}(U)$. ☐

Thus it is clear that all other strategies specified this way define reductions weaker than $\lambda\sigma_w^a$. CBN and CBV are just the the call-by-name and call-by-value strategies of Plotkin (1975). They both use (FVarE) in order to prevent any sharing of subterms.

The next two strategies are like CBN but add sharing in the way used by functional language implementation: CBNeedE is the call-by-need strategy (Launchbury 1993, Ariola et al. 1995); CBNeedG is an adaption of Wadsworth's (1971) "λ-graph reduction" to weak reduction (we return to these below).

The last two strategies realize *n-parallelism with sharing.* The *Specul-*$\|_n$ picks addresses everywhere in the term expecting that some reductions will be useful and the *Spine-*$\|_n$ selects addresses in the head subterm disallowing

(Rap) rule in order to compute the weak head normal form. The simple formulation we have found has promise for proving properties of parallel reduction, however, parallel reductions are otherwise outside the scope of this paper and the rest of this section focuses on sequential reduction.

4.2 "Call-by-Need" strategies

We start by stating some structural properties of terms preserved by sequential reduction. We restrict our attention to just "Call-by-Need-like" strategies. Our intuition about such reduction strategies is that they only reduce a redex in a certain part of a term. This is captured by the following:

Definition 4.4 (Argument local terms). Given T and $a, b \in \mathrm{addr}(T)$, b is the *right subaddress* of a if b occurs only in contexts $C\{(U\ E^b)^a\}$. T is *argument local* if for each address $b \in addr(T)$ which is subaddress of some a then b occurs only in configuration $C'\{(U\ F^b)^a\}$ for the same a.

Definition 4.5 (Head Terms).

Closure Terms are admissible closed addressed terms and defined inductively by

$$P ::= M[s]^a \mid (P M[s]^a)^b \qquad \text{such that } a \notin \mathrm{addr}(P) \qquad \text{(Head Term)}$$
$$s ::= P \cdot s \mid \perp \cdot s \mid \mathrm{id} \qquad \text{(LSubstitution)}$$

We call terms of the form $\_[\_]$ *closures*.

Head Terms are closure terms that are argument local.

The insight is that these term classes capture the essence of sequential and call-by-name/call-by-need reduction. Argument local terms ensure that right arguments of applications are never reduced since they cannot occur in the right argument of an application. In fact, only terms in substitutions are updated.

Proposition 4.6. *Given two terms P and T and a strategy $\mathcal{S}$, such that $P \underset{/\mathcal{S}}{\twoheadrightarrow} T$. If $\mathcal{S}$ disables* (Rap) *and enables* (FVarE) *only when E (in $\underline{0}\,[E^b{\cdot}s]^a$) is a value then if P is a head term then T is a head term.*

Proof. Since T is a head term then it has the form

$$(\cdots(N_0[s_0]^{a_0}\ N_1[s_1]^{a_1})^{b_1}\cdots N_n[s_n]^{a_n})^{b_n}$$

where the s_i's contain only terms of this form, and the a_i do not belong to $addr(N_0[s_0]^{a_0})$. The proof is by case analysis of each rule of $\lambda\sigma_w^a$.

$\mathbf{B}_w$

$$T = C\{((\lambda M)[s]^c\ N[t]^b)^a\} \to C\{M[N[t]^b \cdot s]^a\} = P$$

Since T is a head term, then each occurrence of the subterm $((\lambda M)[s]^c\ N[t]^b)^a$ is not a right argument of an application. Hence replacing this subterm by $M[N[t]^b \cdot s]^a$ preserves head term property.

FvarE

$$T = C\{\,\underline{0}\,[(\lambda M)[s]^b \cdot t]^a\} \to C\{(\lambda M[s]^a\} = P$$

(FvarE) is restricted to values and this obviously preserves head term property. if it is not restricted then one can get non argument local terms.

Apply the same reasoning to the other rules. $\square$

Corollary 4.7. *Given a reduction* $M[\mathrm{id}]^a \underset{\mathcal{S}}{\longmapsto\!\!\!\!\rightarrow} T$ *using a strategy* $\mathcal{S}$. *If* $\mathcal{S}$ *is Call-by-needE, Call-by-NeedG, or some combined Call-by-Need strategy*[3] *then* T *is a head term*

We conclude this section by relating the system we have developed to Ariola et al.'s (1995) "standard call-by-need λ_{let}-calculus" shown in Fig. 5. In λ_{let}, a substitution is represented by nested lets which means that sharing is tied to the term structure. This corresponds to machines with "shared environments" (*i.e.*, using a linked list of frames) hence λ_{let} corresponds to $\lambda\sigma_w^a$ with an environment-based evaluation strategy (the proof is given below). Notice that, strictly speaking, the above rewrite system is *higher order* since $E\{x\}$[4] in the left-hand side of (let_s-V) means that x is the head subterm, *i.e.*, the left-innermost subterm. Indeed, this notation capture the following inductive class of terms

$$E\{x\} ::= x \mid E\{x\}\,M \mid \mathsf{let}\ y = N\ \mathsf{in}\ E\{x\} \qquad (1)$$

[3] We mean a strategy that uses both rules (FvarE) and (FvarG).

[4] The notation $E\{\_\}$ is not a context but an evaluation context, *i.e.*, a window in a term that enables us to decide the redex to reduce.

Syntax.

$$
\begin{array}{rll}
M, N ::= & x \mid V \mid MN \mid \mathsf{let}\ x = M\ \mathsf{in}\ N & \text{(Terms)} \\
V ::= & \lambda x.M & \text{(Values)} \\
A ::= & V \mid \mathsf{let}\ x = M\ \mathsf{in}\ A & \text{(Answers)} \\
E, F ::= & [\,] \mid EM \mid \mathsf{let}\ x = M\ \mathsf{in}\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ F\{x\} & \text{(Eval. Contexts)}
\end{array}
$$

Reduction. $\xrightarrow[\mathsf{let}]{}$ is defined by the rules

$$
\begin{array}{rcll}
(\lambda x.M)N & \to & \mathsf{let}\ x = N\ \mathsf{in}\ M & (\mathsf{let}_s\text{-I}) \\
\mathsf{let}\ x = V\ \mathsf{in}\ E\{x\} & \to & \mathsf{let}\ x = V\ \mathsf{in}\ E\{V\} & (\mathsf{let}_s\text{-V}) \\
(\mathsf{let}\ x = M\ \mathsf{in}\ A)N & \to & \mathsf{let}\ x = M\ \mathsf{in}\ AN & (\mathsf{let}_s\text{-C}) \\
\mathsf{let}\ x = (\mathsf{let}\ y = M\ \mathsf{in}\ A)\ \mathsf{in}\ E & \to & \mathsf{let}\ y = M\ \mathsf{in}\ \mathsf{let}\ x = A\ \mathsf{in}\ E & (\mathsf{let}_s\text{-A})
\end{array}
$$

Figure 5: Ariola et al.'s "standard call-by-need calculus," λ_{let}.

To show the equivalence between our call by need and that of Ariola et al. we have to translate expressions of the λ_{let}-calculus into addressed terms and vice versa. ρ and ρ' are *lists* of variables $(x, y, \cdots)$, where $\rho + \rho'$ is the result of appending ρ and ρ', and $\rho(n)$ is ρ's n'th variable.

Definition 4.8 (translation $\lambda_{\mathsf{let}} \to \lambda\sigma_w^a$). Let ρ and ρ' be *lists* of variables $(x, y, \cdots)$. $\rho + \rho'$ is the result of appending ρ and ρ' and $\rho(n)$ is ρ's n'th variable. $\mathcal{S}\,[\![M]\!]$ means $\mathcal{S}\,[\![M]\!]\ \mathrm{id}\,()$ given by

$\_$	$\mathcal{S}\,[\![\_]\!]\,s\rho$	$\mathcal{S}'\,[\![\_]\!]\,\rho$
x	$(\mathcal{S}'\,[\![x]\!]\,\rho)\,[s]^a$	$\underline{n}$ such that $\rho(n) = x$
$\lambda x.M$	$(\mathcal{S}'\,[\![\lambda x.M]\!]\,\rho)\,[s]^a$	$\lambda(\mathcal{S}'\,[\![M]\!]\,(x \cdot \rho))$
MN	$((\mathcal{S}\,[\![M]\!]\,s\rho)\,(\mathcal{S}'\,[\![N]\!]\,\rho)[s]^a)^b$	$\mathcal{S}'\,[\![M]\!]\,\rho\ \mathcal{S}'\,[\![N]\!]\,\rho$
$\mathsf{let}\ x = N\ \mathsf{in}\ M$	$\mathcal{S}\,[\![M]\!]\,((\mathcal{S}\,[\![N]\!]\,s\rho) \cdot s)\,(x \cdot \rho)$	$(\lambda(\mathcal{S}'\,[\![M]\!]\,(x \cdot \rho)))\,(\mathcal{S}'\,[\![N]\!]\,\rho)$

with a and b fresh everywhere.

Lemma 4.9. *Given a λ_{let}-term M, a $\lambda\sigma_w^a$-substitution s, and an environment ρ. Then there exist two $\lambda\sigma_w^a$-terms N_1 and N_2 such that $N_1 = (\mathcal{S}'\,[\![M]\!]\,\rho)[s]^a$, $N_2 = \mathcal{S}\,[\![M]\!]\ s\,\rho$, and $N_1 \xrightarrow[\lambda\sigma_w^a]{} \!\!\!\!\!\to N_2$.*

Proof. The proof is by induction on the structure of M.

- if $M = \mathsf{let}\ x = N\ \mathsf{in}\ M$ then

$$\begin{aligned}
N_1 &= (\mathcal{S}' [\![\mathsf{let}\ x = N\ \mathsf{in}\ M]\!]\, \rho)[s]^a \\
&= ((\lambda(\mathcal{S}' [\![M]\!]\, (x \cdot \rho)))\, (\mathcal{S}' [\![N]\!]\, \rho))[s]^a \quad \text{by definition} \\
&\xrightarrow[\text{App+B}_w]{} (\mathcal{S}' [\![M]\!]\, (x \cdot \rho))[(\mathcal{S}' [\![N]\!]\, \rho)[s]^b \cdot s]^a \\
&= \mathcal{S} [\![M]\!]\, ((\mathcal{S} [\![N]\!]\, s\rho) \cdot s)\, (x \cdot \rho) \quad \text{by induction hypothesis} \\
&= \mathcal{S} [\![\mathsf{let}\ x = N\ \mathsf{in}\ M]\!]\ s\, \rho \quad \text{by definition} \\
&= N_2
\end{aligned}$$

A similar proof applies to the other cases. $\square$

The main difficulty is to capture the sharing contained in a $\lambda\sigma_w^a$-term expressed by several occurrences of a term labelled with the same address in a let-term. For instance, the term $E^a E^a$ is translated to $\mathsf{let}\ a = [\![E]\!]_{LET}\ \mathsf{in}\ a\, a$.

Definition 4.10 (partial translation $\lambda\sigma_w^a \to \lambda_{\mathsf{let}}$). For an argument local term E^a, $\mathcal{L} [\![E^a]\!]$ means $\mathcal{A}(\mathcal{C} [\![E]\!]\, \varnothing)$ where

$\_$	$\mathcal{A}(M, \_)$	where
$\{E^b\} \cup \mathbf{S}$	$\mathcal{A}(\mathsf{let}\ b = M'\ \mathsf{in}\ M, \mathbf{S})$	$(M', \mathbf{S}') = \mathcal{C} [\![E]\!]\ \mathbf{S}$
$\varnothing$	M	
	$\mathcal{C} [\![\ \_\ ]\!]\ \mathbf{S}$	
$E^a M[s]^b$	$(\ M' (\mathcal{V} [\![M[s]]\!]\ ())\ ,\ \mathbf{S}'\ )$	$(M', \mathbf{S}') = \mathcal{C} [\![E]\!]\ \mathbf{S} \cup \{s\}$
$M[s]$	$(\ \mathcal{V} [\![M[s]]\!]\ ()\ ,\ \mathbf{S} \cup \{s\}\ )$	
	$\mathcal{V} [\![\ \_\ ]\!]\, \rho$	
$M[E^b \cdot s]$	$\mathcal{V} [\![M[s]]\!]\, (\rho + b)$	
$M[\mathrm{id}]$	$\mathcal{V} [\![M]\!]\, \rho$	
λM	$\lambda x.(\mathcal{V} [\![M]\!]\, (x \cdot \rho))$	x a fresh variable
$M N$	$(\mathcal{V} [\![M]\!]\, \rho)\, (\mathcal{V} [\![N]\!]\, \rho)$	
$\underline{n}$	$\rho(n)$	

Lemma 4.11. *If two de Bruijn terms M and N, a $\lambda\sigma_w^a$-substitution s, and an environment ρ, then*

$$\mathcal{V} [\![M N[s]]\!]\, \rho = (\mathcal{V} [\![M[s]]\!]\, \rho)(\mathcal{V} [\![N[s]]\!]\, \rho)$$

Proof. Induction on the structure of s. □

Lemma 4.12 (projections). *The projection diagrams, in Figure 6 are correct.*

$$
\begin{tikzcd}
\bullet \arrow[r, dash, "\mathrm{CBNeedE}"'] \arrow[d, "\mathsf{let}"'] & \bullet \arrow[d, "\mathsf{let}"] \\
\circ \arrow[r, dashed, dash, "\mathsf{let}"'] & \circ
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
\bullet \arrow[r, dash, "\mathsf{let}"'] \arrow[d, "\lambda\sigma_w^a"'] & \bullet \arrow[d, "\lambda\sigma_w^a"] \\
\circ \arrow[r, dashed, dash, "\mathrm{CBNeedE}"'] & \circ
\end{tikzcd}
$$

Figure 6: Soundness and Completeness.

Proof. 1. **projection of λ_{let} in CBNeedE.** The proof is by induction on the structure of the evaluation context E. If the rewrite is at the root, we proceed by case analysis of λ_{let}-rules:

let_s-I

$$
\begin{aligned}
\mathcal{S}\,[\![(\lambda x.M)N]\!]\;\mathrm{id}\,() &= ((\lambda \mathcal{S}'\,[\![M]\!]\;(x\cdot\rho))(\mathcal{S}'\,[\![N]\!]\;())[\mathrm{id}]^c)^a \\
&\xrightarrow[\lambda\sigma_w^a]{} (\mathcal{S}'\,[\![M]\!]\;(x\cdot\rho))[(\mathcal{S}'\,[\![N]\!]\;())[\mathrm{id}]^c\cdot\mathrm{id}]^a \\
&= \mathcal{S}\,[\![M]\!]\;((\mathcal{S}\,[\![N]\!]\;\mathrm{id}\,())\cdot\mathrm{id})\,(x\cdot\rho) \\
&= \mathcal{S}\,[\![\mathsf{let}\ x = N\ \mathsf{in}\ M]\!]\;\mathrm{id}\,()
\end{aligned}
$$

The other cases are similar and the most tedious case (let_s-V) requires an induction on the evaluation context $E\{x\}$ to distribute the substitution through the term.

If $E = E_1 M$ such that $E_1 \xrightarrow[\lambda_{\mathsf{let}}]{} E_2$ then by the induction hypothesis $\mathcal{S}\,[\![E_1]\!]\;\mathrm{id}\,() \xrightarrow[\mathrm{CBNeedE}]{} \!\!\!\!\!\!\!\!\rightarrow \mathcal{S}\,[\![E_2]\!]\;\mathrm{id}\,()$ holds.

$$
\begin{aligned}
\mathcal{S}\,[\![E_1M]\!]\;\mathrm{id}\,() &= ((\mathcal{S}\,[\![E_1]\!]\;\mathrm{id}\,())(\mathcal{S}'\,[\![M]\!]\;())[\mathrm{id}]^b)^a \\
&\xrightarrow[\mathrm{CBNeedE}]{} \!\!\!\!\!\!\!\!\rightarrow ((\mathcal{S}\,[\![E_2]\!]\;\mathrm{id}\,())(\mathcal{S}'\,[\![M]\!]\;())[\mathrm{id}]^b)^a \\
&= \mathcal{S}\,[\![E_2M]\!]\;\mathrm{id}\,()
\end{aligned}
$$

If $E = \mathsf{let}\ x = E_1\ \mathsf{in}\ F\{x\}$ such that $E_1 \xrightarrow[\lambda_{\mathsf{let}}]{} E_2$ and by induction hypothesis $\mathcal{S}[\![E_1]\!]\,\mathrm{id}\,() \xrightarrow[\mathrm{CBNeedE}]{} \mathcal{S}[\![E_2]\!]\,\mathrm{id}\,()$

$$\mathcal{S}[\![E]\!]\,\mathrm{id}\,() = \mathcal{S}[\![F\{x\}]\!]\,((\mathcal{S}[\![E]\!]\,\mathrm{id}\,())\cdot\mathrm{id})\,(x\cdot())$$

then by induction on $F\{x\}$, we can show that it is a CBNeedE rewrite.

2. **Projection of CBNeedE to λ_{let}.** The proof is by induction on the structure of the head term P following the condition of CBNeedE. If the rewrite is at the root, we proceed by case analysis of $\lambda\sigma_w^a$-rules:

App

$$\begin{aligned}
\mathcal{L}[\![MN[s]^a]\!] &= \mathcal{A}(\mathcal{C}[\![MN[s]]\!]\,\varnothing) \\
&= \mathcal{A}(\mathcal{V}[\![MN[s]]\!]\,(), s) \\
&= \mathcal{A}(\mathcal{V}[\![M[s]]\!]\,()\,\mathcal{V}[\![N[s]]\!]\,(), s) \quad \text{by lemma 4.11} \\
&= \mathcal{A}(\mathcal{C}[\![M[s]^bN[s]^c]\!]\,\varnothing) \quad b \text{ and } c \text{ are fresh addresses} \\
&= \mathcal{L}[\![(M[s]^bN[s]^c)^a]\!]
\end{aligned}$$

The other cases are similar.

If $P = (Q\,M[s]^b)^a$ such that $Q \xrightarrow[\mathrm{CBNeedE}]{} Q'$ and by induction hypothesis $D : \mathcal{L}[\![Q]\!] \xrightarrow[\lambda_{\mathsf{let}}]{=} \mathcal{L}[\![Q']\!]$. Notice that, in general, rewriting with CBNeedE the term P gives $(Q'M[s']^b)^a$ and not $(Q'M[s]^b)^a$.

$$\begin{aligned}
\mathcal{L}[\![P]\!] &= \mathcal{A}(\mathcal{C}[\![(QM[s]^b)^a]\!]\,\varnothing) \\
&= \mathcal{A}(M'(\mathcal{V}[\![M[s]]\!]\,()), S') \\
&\quad \text{such that } (M', S') = \mathcal{C}[\![Q]\!]\,s \\
&\xrightarrow[\lambda_{\mathsf{let}}]{=} \mathcal{A}(M''(\mathcal{V}[\![M[s]]\!]\,()), S'') \quad \text{using the same derivation } D \\
&= \mathcal{A}(\mathcal{C}[\![(Q'M[s']^b)^a]\!]\,\varnothing) \\
&= \mathcal{L}[\![(Q'M[s']^b)^a]\!]
\end{aligned}$$

If $P = \underline{0}\,[Q\cdot s]^a$ and $Q \xrightarrow[\mathrm{CBNeedE}]{} Q'$ and by induction hypothesis $D' : \mathcal{L}[\![Q]\!] \xrightarrow[\lambda_{\mathsf{let}}]{=} \mathcal{L}[\![Q']\!]$. Notice that, in general, rewriting with CBNeedE the term P gives $\underline{0}\,[Q' \cdot s']^a$ and not $\underline{0}\,[Q' \cdot s]^a$.

$$\begin{aligned}
\mathcal{L}\,[\![P]\!] &= \mathcal{A}(\mathcal{C}\,[\![\,\underline{0}\,[Q \cdot s]]\!]\,\varnothing) \\
&= \mathcal{A}(\mathcal{V}\,[\![\,\underline{0}\,[Q \cdot s]]\!]\,(), Q \cdot s) \\
&= \mathcal{A}(b, E^b \cdot s) \quad Q = E^b \\
&\mathcal{A}(\mathsf{let}\ b = M\ \mathsf{in}\ b, S) \quad \text{where } (M, S) = \mathcal{C}\,[\![E]\!]\,s \\
&\xrightarrow[\lambda_{\mathsf{let}}]{=} \mathcal{A}(\mathsf{let}\ b = M'\ \mathsf{in}\ b, S') \quad \text{using the same derivation } D' \\
&= \mathcal{A}(\mathcal{C}\,[\![\,\underline{0}\,[Q' \cdot s']]\!]\,\varnothing \\
&= \mathcal{L}\,[\![\,\underline{0}\,[Q' \cdot s']^a]\!]
\end{aligned}$$

□

The diagrams in Figure 6 do not express a one-to-one correspondence among computation steps. This comes from the way substitutions are performed in the two calculi. $\lambda\sigma_w^a$ works at a more primitive level w.r.t. substitutions: it is a "small step" calculus in the sense that basic and constant-time operations are implemented. This sometimes requires distribution of the substitution of the evaluation context $E\{x\}$ to get to the redex to reduce whereas in λ_{let} this action is performed in one step.

But, the λ_{let}-calculus is primitive enough to reason (equationally) about properties of call-by-need strategy.

Proposition 4.13. *CBNeedE is sound and complete w.r.t.* λ_{let}.

Proof. A way to convince ourselves of the soundness and completeness of the two calculi is to compare the number of B_w-redex and let_s-I-redex (which are $\beta - redex$) starting with a λ-term. By the previous lemma we can notice that there are one to one correspondence between these two rules, and (B_w) is never used to project another rule. □

5 Constructors and Recursion

In this section, we deal with two important features of functional programming languages, namely *algebraic data structures* in the form of constructors that can be investigated using a 'case' selector statement, and *recursion* in the form of an explicit fixed point operator.

5.1 Algebraic data structure

Definition 5.1. $\lambda\sigma c_w^a$ is the system obtained by extending the definition of $\lambda\sigma_w^a$ (pure) preterms to include

$$M, N ::= \cdots \mid C_i \mid \langle C_1 : M_1, \ldots, C_m : M_m \rangle$$

where the C_i range over some finite set of constructors of fixed arity, $\mathrm{ar}(C_i)$, such that $\mathrm{fi}(C_i) = \{0, \ldots, \mathrm{ar}(C_i)-1\}$. The rule (Case) is a kind of application defined by

$$(C_i[\vec{T} \cdot s]^b \ \langle \vec{C} : \vec{N} \rangle \, [t]^c)^a \to N_i[\vec{T} \cdot t]^a \qquad \text{(Case)}$$

with $\vec{T} \cdot s = T_1 \cdots T_{\mathrm{ar}(C_i)} \cdot s_1 \cdot s_2 \cdot \ldots$ and $\langle \vec{C} : \vec{N} \rangle = \langle C_1 : N_1, \ldots, C_n : N_n \rangle$.

The definition highlights that the only reduction involving data is to select an entry of a case argument corresponding to which particular C_i it was built with. Also addresses are never allowed inside constructions in accordance with the tradition that constructed objects are considered as *values* similar to abstractions and hence no strategy should be allowed to reduce "inside."

Example 5.2. Suppose we have two constructors Z of arity 0 and S of arity 1. Consider the reduction of the term $(\lambda\ \underline{0}\ \langle \mathsf{Z} : \mathsf{I}, \mathsf{S} : \mathsf{K} \rangle)\mathsf{Z}$.

$$\begin{aligned}
((\lambda\ \underline{0}\ \langle \mathsf{Z} : \mathsf{I}, \mathsf{S} : \mathsf{K} \rangle)\mathsf{Z})[id]^a &\to ((\lambda\ \underline{0}\ \langle \mathsf{Z} : \mathsf{I}, \mathsf{S} : \mathsf{K} \rangle)[id]^b \mathsf{Z}[id]^c)^a \\
&\to (\ \underline{0}\ \langle \mathsf{Z} : \mathsf{I}, \mathsf{S} : \mathsf{K} \rangle)[\mathsf{Z}[id]^c \cdot id]^a \\
&\to (\ \underline{0}\,[\mathsf{Z}[id]^c \cdot id]^d \,\langle \mathsf{Z} : \mathsf{I}, \mathsf{S} : \mathsf{K} \rangle\ [\mathsf{Z}[id]^c \cdot id]^e)^a \\
&\to (\mathsf{Z}[id]^c \,\langle \mathsf{Z} : \mathsf{I}, \mathsf{S} : \mathsf{K} \rangle\ [\mathsf{Z}[id]^c \cdot id]^e)^a \\
&\to \mathsf{I}[\mathsf{Z}[id]^c \cdot id]^a
\end{aligned}$$

5.2 Recursive code and data

Recursion is somewhat more involved. $\lambda\sigma c\mu_w^a$ denotes the easy solution, which is to reduce terms of the form μM by unfolding with

$$\mu M[s]^a \to M[\mu M[s]^a \cdot s]^b \qquad b \text{ fresh} \qquad \text{(Unfold)}$$

(Unfold) must of course be applied lazily to avoid infinite unfolding.

Another solution consists in delaying unfolding until needed. The trick is to augment the explicit "horizontal" sharing that we have introduced in previous sections through addresses with explicit "vertical" sharing (using the terminology of Ariola & Klop 1994). We have chosen to do this using the $\bullet^a$ "backpointer" syntax (Felleisen & Friedman 1989, Rose 1996): reducing a fixed point operator places a $\bullet$ at the location where unfolding should happen when (if) it is needed.

The difference is illustrated by Fig. 7. Consider the initial term with a

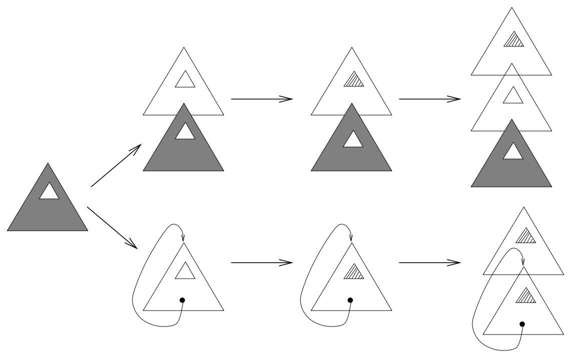

Figure 7: A recursive redex occurrence.

large (shaded) μ-redex containing a smaller (white) redex. Now, we wish to reduce the outer and then the inner redex. The top reduction shows what happens with (Unfold): the redex is duplicated before it is reduced. In contrast using an explicit backpointer, illustrated at the bottom, makes it possible to share the redex by delaying the unfolding until the reduction has happened. The price to pay is that all redexes of a term are no longer present in any of its representation, since backpointers can block specific redexes. Moreover, the admissibility definition becomes slightly more complicated and hence its preservation after a rewriting with $\lambda\sigma c_w^a$-rules has to be addressed carefully.

Definition 5.3 (cyclic addressing). Cyclic addressing is the following generalization of Def. 3.8; assume a set of addressed terms $\mathbf{T}$:

1. The *cyclic addressed preterms*: $\mathbf{T}^\bullet$ allow subterms of the form $\bullet^a$ wherever an address is otherwise allowed.

2. t is *graphic* if all addresses $a \in \operatorname{addr}(t)$ satisfy either $t@a = \{\bullet^a\}$, or

$t@a = \{s^a\}$ where $s@a \subseteq \{\bullet^a\}$. Admissible preterms are called *terms*; terms without $\bullet$ are *acyclic*.

3. Given a rewrite relation $\rightarrow$ on (acyclic terms) $\mathbf{T}$. The *cyclic extension*, $\circ\!\!\rightarrow$, is the rewrite relation defined as follows. Assume $t \rightarrow u$. For each backpointer where admissibility is violated: unfold the original definition for each possible C with $u = C\{\bullet^a\}$ and $a \notin \mathrm{addr}(C\{\,\})$, *i.e.*, replace $C\{\bullet^a\}$ with $C\{t@a\}$.

Notice that the cyclic extension of a set of rules can be derived by inserting explicit unfolding *where an address is removed.*

Definition 5.4 ($\lambda\sigma c\mu_w^{a\bullet}$)**.**

1. $\lambda\sigma c\mu_w^{a\bullet}$*-terms* are cyclic addressed $\lambda\sigma c_w^a$-terms extended with recursion terms:

$$M, N ::= \cdots \;\big|\; \mu M$$

2. $\lambda\sigma c\mu_w^{a\bullet}$*-reduction*, $\circ\!\!\twoheadrightarrow$, is the cyclic extension of $\lambda\sigma c_w^a$-reduction and the rule

$$\mu M[s]^a \rightarrow M[\bullet^a \cdot s]^a \tag{Cycle}$$

Let us briefly consider what "cyclic extension" means. It is clear that (Cycle) itself preserves (cyclic) admissibility but not all the other rules do. Fortunately we can systematically reformulate the rules in order to insert the unfoldings which are needed explicitly. If we write $T\{\bullet^a := U\}$ for the operation which replaces each $\bullet^a$ in T by U, then the principle is simple: whenever an address a is *removed* from a subterm t then all occurrences of $\bullet^a$ *inside* t must be unfolded to the value of the entire subterm. This affects the $\lambda\sigma_w^a$-rules from Fig. 3 as follows:

$$((\lambda M)[s]^b\, U)^a \rightarrow M[U \cdot (s\{\bullet^b := \lambda M[s]^b\})]^a \tag{$\mathrm{B}_w^\bullet$}$$

$$\underline{0}\,[E^b \cdot s]^a \rightarrow (E\{\bullet^a := \underline{0}\,[E^b \cdot s]^a\})^b \tag{FVarG$^\bullet$}$$

$$\underline{0}\,[E^b \cdot s]^a \rightarrow (E\{\bullet^b := E^b\})^a \tag{FVarE$^\bullet$}$$

and (Case) from above changes as follows:

$$(C_i[\vec{T} \cdot s]^b\ \big\langle \vec{C} : \vec{N} \big\rangle\, [t]^c)^a \rightarrow N_i[\vec{T'} \cdot t']^a \tag{Case$^\bullet$}$$

with $\vec{T} = T_1 \cdot \ldots \cdot T_{\mathrm{ar}(C_i)}$, $\vec{T'} = \vec{T}\{\bullet^b := C_i[\vec{T} \cdot s]^b\}$, $t' = t\{\bullet^c := \left\langle \vec{C} : \vec{N} \right\rangle [t]^c\}$.

We are going to prove that *cycling rewriting* implies *unfolding rewriting*. To do that we use an intermediate rewriting which we call *cycling rewriting with history* whose main idea is to keep track of substitutions of $\bullet$ along a cycling rewriting. A sequence of such $\bullet$ assignments is called a history.

Definition 5.5 (History). A history is a sequence ρ of the form

$$\{\bullet^{a_1} := \mu M_1[s_1], \ldots, \bullet^{a_n} := \mu M_n[s_n]\}$$

Assigning a history ρ to an addressed term P yields a term $P\rho$ that substitutes all $\bullet$ of ρ by their corresponding terms addressed with fresh addresses. A formal definition is left to the reader.

Definition 5.6 (Cycling rewriting with history). Cycling rewriting with history $\lambda\sigma c\mu_w^{a\bullet}\rho$ rewrites a pair of a term and a history (P, ρ) to a similar pair (Q, π) such that $P \underset{\lambda\sigma c\mu_w^{a\bullet}}{\multimap\!\!\!\longrightarrow} Q$ and $\rho = \pi$ except when the rewrite from P to Q is a (Cycle) $\mu M[s]^a \to M[\bullet^a \cdot s]^a$ whereas $\pi = \{\bullet^a := \mu M[s]\rho\} \cup \rho$.

Lemma 5.7. *Given an addressed term $P \in \lambda\sigma c\mu_w^a$-term*[5]*. If $(P, \{\}) \underset{\lambda\sigma c\mu_w^{a\bullet}\rho}{\twoheadrightarrow} (Q, \rho)$ then $Q\rho \in \lambda\sigma c\mu_w^a$, i.e., Q contains no terms of the form $\bullet^a$.*

Proof. Obvious. □

Lemma 5.8. *If $P \underset{\lambda\sigma c\mu_w^{a\bullet}}{\multimap\!\!\!\longrightarrow} Q$ then ρ and ρ' exist such that $(P, \rho) \underset{\lambda\sigma c\mu_w^{a\bullet}\rho}{\multimap\!\!\!\longrightarrow} (Q, \rho')$.*

Proof. The proof is by analysis of each rule of $\lambda\sigma c\mu_w^{a\bullet}$. Assume $\rho = \{\}$. □

Lemma 5.9. *If $D : (P, \{\}) \underset{\lambda\sigma c\mu_w^{a\bullet}\rho}{\multimap\!\!\!\twoheadrightarrow} (Q, \rho)$ such that P is a $\lambda\sigma c\mu_w^a$-term, then $P \underset{\lambda\sigma c\mu_w^a}{\vdash\!\!\!\twoheadrightarrow} Q'\rho$ where Q' is an address renaming*[6] *of Q.*

Proof. The proof is by induction on the length of D (denoted by $|D|$).

- If $|D| = 1$ then by induction on the structure of P. If the rewrite is at the root the proof is by case analysis on the possible rule of $\lambda\sigma c\mu_w^{a\bullet}\rho$. For instance, consider that P is B_w redex and since $P = (\lambda M[s]^b T)^a$ is $\lambda\sigma c\mu_w^a$-term then $\lambda M[s]^b$ contains no $\bullet^b$ subterms.

[5] The important point here is that P contains no $\bullet^a$.

[6] The renaming doesn't affect terms of the form $\bullet^a$.

- If $|D| = n+1$ then $D = (P, \{\}) \xrightarrow[\lambda\sigma c\mu_w^{a\bullet}\rho]{n} (P', \rho') \xrightarrow[\lambda\sigma c\mu_w^{a\bullet}\rho]{} (Q, \rho)$ and by induction hypothesis $P \twoheadrightarrow_{\lambda\sigma c\mu_w^a} P''\rho'$ where P'' is an address renaming of P'. We proceed by induction on the structure of P'. If the rewrite is at the root then one has to analyze each rule of $\lambda\sigma c\mu_w^{a\bullet}\rho$.

  Cycle $P' = \mu M[s]^a$

$$(\mu M[s]^a, \rho) \xrightarrow[\text{Cycle}]{} (M[\bullet^a \cdot s]^a, \{\bullet^a := \mu M[s]\rho\} \cup \rho)$$

$$\mu M[s]^a\rho \xrightarrow[\text{Unfold}]{} M[\mu M[s]^b]^a\rho \quad b \text{ fresh address}$$

$$= M[\bullet^a \cdot s]^a\{\bullet^a := \mu M[s]\rho\} \cup \rho$$

  $\mathbf{B}_w^\bullet$ $P' = (\lambda M[s]^b T)^a$. Two cases are possible:
  If $\lambda M[s]^b$ contains occurences of $\bullet^b$ then $\rho = \{\bullet^b := \mu M'[s']\} \cup \rho'$ and we have

$$(R, \{\}) \twoheadrightarrow_{\lambda\sigma c\mu_w^{a\bullet}\rho} (C\{\mu M'[s'']^b\}, \rho'') \twoheadrightarrow_{\lambda\sigma c\mu_w^{a\bullet}\rho} ((\lambda M[s]^b T)^a, \rho)$$

  such that $\mu M'[s'']^b\rho'' = \mu M'[s']^b$. Then there exists a reduction $(\mu M'[s']^b, \{\}) \xrightarrow[\lambda\sigma c\mu_w^{a\bullet}\rho]{m} (\lambda M[s]^b, \rho)$ and $m \leq n$. By induction hypothesis $\mu M'[s']^b \twoheadrightarrow_{\lambda\sigma c\mu_w^a} \lambda M[s_3]^b\rho$ where s_3 is an address renaming of s.

$$(\lambda M[s]^b T)^a, \rho) \xrightarrow[B_w^\bullet\rho]{} (M[T \cdot (s\{\bullet^b := \lambda M[s]^b\})]^a, (\{\bullet^b := \mu M'[s']\} \cup \rho'))$$

$$\begin{aligned}
(\lambda M[s]^b T)^a\rho &= (\lambda M[s\{\mu M'[s']^c\}]^b T)^a\rho' \quad c \text{ fresh} \\
&\xrightarrow[B_w]{} M[T \cdot s\{\mu M'[s']^c\}]^a\rho' \\
&\xrightarrow[\text{Unfold}]{} M[T \cdot s\{M'[\mu M'[s']^d \cdot s']^c\}]^a\rho' \quad d \text{ fresh} \\
&\twoheadrightarrow_{\lambda\sigma c\mu_w^a} M[T \cdot s\{\lambda M[s'']^c\}]^a\rho' \quad \text{where } s'' = s_3\{\bullet^b := \mu M'[s']^d\} \\
&= M[T \cdot s\{\lambda M[s_3]^c\}]^a(\{\bullet^b := \mu M'[s']\} \cup \rho') \\
&\approx M[T \cdot s\{\lambda M[s]^b\}]^a(\{\bullet^b := \mu M'[s']\} \cup \rho') \\
&\qquad \text{since } b \text{ is fresh w.r.t the term } M[T \cdot s\{\lambda M[s'']^c\}]^a\rho'
\end{aligned}$$

  (Unfold) introduces a new address whereas $(\mathrm{B}_w^\bullet)$[7] reuse the address b since it becomes fresh after a $\mathrm{B}_w^\bullet$ rewrite step. In fact, $\mathrm{B}_w^\bullet$

[7] Also (FVarE$^\bullet$), (FVarG$^\bullet$), and (Case$^\bullet$) reuses addresses when they make the unfolding.

performs two tasks namely to reduce the β-redex and to *unfold* the recursive function of this redex. This explains why a renaming of addresses is necessary.

The other cases are similar to the two previous ones. $\square$

Theorem 5.10. *If $P \underset{\lambda\sigma c\mu_w^{a\bullet}}{\circ\!\!\twoheadrightarrow} Q$ such that $P$is a $\lambda\sigma c\mu_w^a$-term then ρ exists such that $P \underset{\lambda\sigma c\mu_w^a}{\mapsto\!\!\twoheadrightarrow} Q'\rho$, where Q' is an address renaming of Q.*

Proof. Combine Lemmas 5.8 and 5.9. $\square$

6 "Trim:" A Space leak free calculus

In this section, we present the Trim-calculus. Its main point is to (Collect) just what is necessary to preserve garbage-freeness (a concept we introduce). We first introduce the general calculus, then we study its restrictions to the call-by-need strategy using environments, finally we use this to show that the STG (Spineless Tagless Graph-reduction) machine of Peyton Jones (1992) does not leak space.

Definition 6.1. *Garbage-free* terms are characterized by

$$\begin{aligned} T_f &::= M[s_f]^a \mid (T_f T_f)^a \mid n^a \mid \bot \qquad \text{with } s_f = s_f|_{fi(M)} \\ s_f &::= \mathrm{id} \mid T_f \cdot s_f \end{aligned}$$

Proposition 6.2. *If $T \xmapsto[\mathrm{C}]{\mathrm{addr}(T)} U$, then U is a garbage-free term.*

Proof. The rule (Collect) collects all unreachable subterms. $\square$

Space leak freeness (precisely formulated for a lazy abstract machine by Sestoft 1997) means that every (Collect)-redex will eventually disappear in a computation. In other words, no unreachable subterm stays indefinitely.

Definition 6.3 (space leak free reduction). Let D be a $\lambda\sigma_w^a$-reduction path starting at a garbage-free term T_0:

$$D : T_0 \xmapsto{I_1} T_1 \xmapsto{I_2} T_2 \xmapsto{I_3} \cdots$$

D is *space leak free* if when $T_n = C\{E^b\}$ such that E^b is garbage then there exists $m > n$ such that E^b is collected in T_m. A reduction relation $\xrightarrow[R]{}$ (of $\lambda\sigma_w^a$-calculus) is space leak free if all its reduction paths starting at a garbage-free term are space leak free.

$\mapsto$ is not a space leak free reduction. A naive way to provide space leak reductions is to normalize w.r.t. (Collect) after several steps of $\mathrm{B}_w + \sigma$ rewriting. This corresponds to *garbage collection.*

Proposition 6.4. $\underset{\mathrm{B}_w+\sigma}{\mapsto\!\!\twoheadrightarrow} \cdot \underset{\mathrm{C}}{\longmapsto}$ *is space leak free.*

Proof. By proposition 6.2, it is clear that after each reduction step, there remain no addressed subterms which are garbage. $\square$

When analyzing the rules of $\lambda\sigma_w^a$, one notices that only (B_w) and (App) produce unreachable terms: (App) introduces two new subaddressed terms of the form $\_[\_]^a$ (closures) on which an application of (Collect) might be necessary, and (B_w) adds a new addressed term (argument) to the function's substitution. If we know that the B_w-redex is garbage free then it is necessary to check only whether the argument is reachable or not. The rule (Case) can also introduce unreachable terms in the substitution $T_1 \cdot \ldots \cdot T_m \cdot t$. Hence, we need to apply (Collect). Thus, we replace these rules with the following:

$$(\lambda M[s]^b T)^a \xrightarrow{a} \begin{cases} M[\perp \cdot s] & \underline{0} \notin \mathrm{fi}(M) \\ M[T \cdot s] & \underline{0} \in \mathrm{fi}(M) \end{cases} \tag{TB_w}$$

$$(MN)[s]^a \xrightarrow{a} (M[s|_{\mathrm{fi}(M)}]^b N[s|_{\mathrm{fi}(N)}]^c)^a \quad b, c \text{ fresh} \tag{TApp}$$

$$(C_i[\vec{T}]^b \ \langle C_1 : N_1, \ldots, C_m : N_m\rangle \, [t]^c)^a \xrightarrow{a} N_i[(\vec{T} \cdot t)|_{\mathrm{fi}(N_i)}] \tag{TCase}$$

With $\vec{T} = T_1 \cdot \ldots \cdot T_{\mathrm{ar}(C_i)}$. Note that none of the recursion rules (Unfold) and (Cycle) introduce space leaks.

Definition 6.5. The Trim-*calculus* is the calculus over addressed terms combining the original (FVarE), (FVarG) and (RVar), with the new (TB_w), (TApp), and (TCase). We write this reduction $\underset{T}{\longmapsto}$.

Remark 6.6. Notice that $\overset{a}{\underset{(\mathrm{TApp})}{\longmapsto}} = \overset{a}{\underset{(\mathrm{App})}{\longmapsto}} \cdot \overset{b}{\underset{\mathrm{C}}{\longrightarrow}} \cdot \overset{c}{\underset{\mathrm{C}}{\longrightarrow}}$, and $\overset{a}{\underset{(\mathrm{TCase})}{\longmapsto}} = \overset{a}{\underset{(\mathrm{Case})}{\longmapsto}} \cdot \overset{a,d_1,\ldots d_m}{\underset{\mathrm{C}}{\longmapsto}}$ where the d_i are the addresses of the T_i.

Theorem 6.7 (Preservation). *Let T be a garbage-free term. If $T \underset{T}{\longmapsto} U$ then U is a garbage-free term.*

Proof. By case analysis of each rule of $\underset{T}{\longmapsto}$. $\square$

Corollary 6.8. Trim *is space leak free.*

Since we cannot rewrite in useless terms, we can claim that Trim is isomorphic modulo substitution under λ to the weak version of Wadsworth's (1971) "λ-graph reduction." The Trim-calculus ensures that all its strategies are space leak free. However, we remind the reader that collecting incurs an overhead, so one has to minimize this task.

In the remainder of this section, we study optimization of call-by-need w.r.t. space leak freeness, in particular we show that STG does not leak space. One feature of call-by-need is that it always selects the leftmost outermost redex. If we apply (App) to the term $C\{(MN)[s]\}$ then we know that the left term (namely, $M[s]$) will be evaluated first. Hence trimming this term is unnecessary w.r.t. space leaking. Similarly, a trimmed version of (B_w) and (Case) are not necessary. The trimmed (App) for call by need becomes

$$(MN)[s]^a \xrightarrow{a} (M[s]^b N[s|_{\mathrm{fi}(N)}]^c)^a \quad b, c \text{ fresh} \tag{TAppN}$$

Replacing (App) of the $\lambda\sigma_w^a$-calculus by (TAppN) forms $\mathrm{Trim_N}$.

Theorem 6.9. $\mathrm{Trim_N}$ *is space leak free for call by name (CBN) and call by need CBNeedE and CBNeedG.*

Proof. $\mathrm{Trim_N}$ trims all terms where computations is postponed, namely, right argument of applications. □

The STG-language, described in Figure 8, is a subset of λ-calculus[8] enriched with *local definitions* (let-expressions), *constructors* including *built-in functions* and *literals* (constants), *selection instruction* (case), and *explicit recursion* (letrec). It is not difficult to define a map from the STG-language to our $\lambda\sigma c\mu_w^{a\bullet}$-terms. The STG-machine uses *update technique*, *i.e.*, updates the argument in the heap after its evaluation for further use, and hence the strategy underlining the STG-machine is *Call-by-NeedE*. As we can see in Figure 8, the rule (let) creates closures by assigning the environment ρ to the bounded expressions N_i's and stores them in the heap. Each environment of N_i is trimmed to its free variables. This operation is expressed in the Figure8 by $(\rho\, vs_i)$. We conclude that the STG-machine is a *duplicated environment machine*.

[8]This subset is called *restricted λ-calculus*, it imposes that the second argument of applications are made of variables only

Syntax. The language of the STG is defined inductively by

$$
\begin{array}{llr}
e ::= & \mathrm{let}\ (var = vs_f \setminus \pi\, vs_b \rightarrow e)^+\ \mathrm{in}\ e & \text{(Local definition)} \\
 & \mid \mathrm{letrec}\ (var = vs_f \setminus \pi\, vs_b \rightarrow e)^+\ \mathrm{in}\ e & \text{(Local recursion)} \\
 & \mid \mathrm{case}\ e\ \mathrm{of}\ ((const\, vs \mid var \mid literal) \rightarrow e)^+ & \text{(Case expression)} \\
 & \mid var\ (var \mid literal)^* & \text{(Application)} \\
 & \mid const\ (var \mid literal)^* & \text{(Saturated constructor)} \\
 & \mid prim\ (var \mid literal)^* & \text{(Saturated Built-in op)} \\
 & \mid literal &
\end{array}
$$

where the'vs_f and the'vs_b are the free and bound variables of e. π is mark to determine if the expression has been updated.

State. contains seven components of four kinds:

Code: as described above. *Environments*: There are two environments: the *local* ρ and *global* σ. They are defined by a map from *variables* to *addresses* in the heap. *Heap*: h, is defined by a map from *addresses* to *closures*, defined by a couple of *code* and *local environment*, *Stacks*: There are three kind of stacks: the *argument stack as*, the *return stack rs* that contains continuations, and the *update stack us*

Evaluation. We present just the rule for evaluating let as this is the main one for us (other rules are described by Peyton Jones 1992).

$$
\begin{array}{lllllll}
 & Eval \left(\begin{array}{ll} \mathrm{let} & x_1 = vs_1 \setminus \pi_1\, xs_1 \rightarrow e_1 \\ & \dots \\ & x_n = vs_n \setminus \pi_n\, xs_n \rightarrow e_n \\ \mathrm{in} & e \end{array} \right) & \rho & as & rs & us & h & \sigma \\
\longrightarrow & Eval\ e & \rho' & as & rs & us & h' & \sigma
\end{array}
$$

$$
\begin{array}{rl}
\text{where } \rho' & = \rho\, [x_1 \mapsto Addr\ a_1, \dots, x_n \mapsto Addr\ a_n] \\
h' & = h \left[\begin{array}{l} a_1 \mapsto (vs_1 \setminus \pi_1\, xs_1 \rightarrow e_1)(\rho_{rhs}\ vs_1) \\ \dots \\ a_n \mapsto (vs_n \setminus \pi_n\, xs_n \rightarrow e_n)(\rho_{rhs}\ vs_n) \end{array} \right] \\
\rho_{rhs} & = \rho
\end{array}
$$

For the letrec rule, replace ρ_{rhs} by ρ' instead of ρ.

Figure 8: Part of the STG-machine

Corollary 6.10. *Peyton Jones's (1992) STG-machine does not leak space.*

Proof. In the STG-language, the arguments of applications and constructors must be variables which are bound by a *let expression.* Consider the term $MN_1 \dots N_n$ which is written $\mathsf{let}\ x_1 = N_1, \dots x_n = N_n\ \mathsf{in}\ Mx_1 \dots x_n$ where $N_1, \dots, N_n$ are annotated with their free variables. Hence, it suffices to trim the environment in let expressions w.r.t free variable of the N_i's which is done by the let-rule of the STG-machine (see figure 8).

□

7 Conclusions

We have studied calculi of weak reduction with sharing, and proposed original solutions to several problems well-known from implementations, *e.g.*, understanding the consequences of sharing and cycles on correctness, proving an adequate model for call-by-need, space leaking, and on a generic implementation design.

Acknowledgements. The authors would like to thank Eva Rose and the anonymous referees for useful suggestions to the manuscript. Finally, the third author is grateful to INRIA Lorraine for funding while this work was undertaken.

References

Abadi, M., Cardelli, L., Curien, P.-L. & Lévy, J.-J. (1991), 'Explicit substitutions', *Journ. Funct. Progr.* **1**(4), 375–416.

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M. & Wadler, P. (1995), A call-by-need lambda calculus, *in* '22nd Principles of Programming Languages', San Francisco, California, pp. 233–246.

Ariola, Z. M. & Klop, J. W. (1994), Cyclic lambda graph rewriting, *in* 'Logic in Computer Science', IEEE Computer Society Press, Paris, France, pp. 416–425.

Bloo, R. & Rose, K. H. (1995), Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection, *in* 'CSN

'95 – Computer Science in the Netherlands', pp. 62–72. ⟨URL:*ftp://ftp.diku.dk/diku/semantics/papers/D-246.ps*⟩

Cardelli, L. (1983), The functional abstract machine, Technical Report TR-107, Bell Labs.

Church, A. (1936), 'An unsolvable problem of elementary number theory', *Amer. J. Math.* **39**, 472–482.

Curien, P.-L. (1983), Combinateurs catégoriques, algorithmes séquentiels et programmation applicative, Thèse de Doctorat d'Etat, Université Paris 7.

Curien, P.-L. (1991), 'An abstract framework for environment machines', *Theor. Comp. Sci.* **82**, 389–402.

Curien, P.-L., Hardin, T. & Lévy, J.-J. (1992), Confluence properties of weak and strong calculi of explicit substitutions, Rapport de Recherche 1617, INRIA. To appear in Journ. ACM.

Curry, H. (1930), 'Grundlagen der kombinatorischen logik', *American Journal of Mathematics* **52**, 509–536, 789–834.

Curry, H. B. & Feys (1958), *Combinatory Logic*, Vol. 1, Elsevier Science Publishers B. V. (North-Holland), Amsterdam.

de Bruijn, N. G. (1972), 'Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem', *Proc. Koninkl. Nederl. Akademie van Wetenschappen* **75**(5), 381–392.

Felleisen, M. & Friedman, D. P. (1989), 'A syntactic theory of sequential state', *Theor. Comp. Sci.* **69**, 243–287.

Gödel, K. (1931), 'Über formal unentscheidbare Sätze der Principia Mathemathica und verwandter Systeme I', *Monatsh für Math. u Phys.* **12**(XXXVIII), 173–198.

Henderson, P. (1980), *Functional Programming—Application and Implementation*, Prentice-Hall.

Huet, G. & Lévy, J.-J. (1991), Computations in orthogonal rewriting systems, II, *in* J.-L. Lassez & G. Plotkin, eds, 'Computational Logic', The MIT press, chapter 12, pp. 415–443.

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

Klop, J. W. (1992), Term rewriting systems, *in* S. Abramsky, D. M. Gabbay & T. S. E. Maibaum, eds, 'Handbook of Logic in Computer Science', Vol. 2, Oxford University Press, pp. 1–116.

Landin, P. J. (1965), 'A correspondance between ALGOL 60 and Church's lambda notation', *Comm. ACM* **8**, 89–101 and 158–165.

Launchbury, J. (1993), A natural semantics for lazy evaluation, *in* '20th Principles of Programming Languages', pp. 144–154.

Maranget, L. (1991), Optimal derivations in weak lambda calculi and in orthogonal rewriting systems, *in* '18th Principles of Programming Languages', pp. 255–268.

McCarthy, J. (1960), 'Recursive functions of symbolic expressions', *Comm. ACM* **3**(4), 184–195.

Peyton Jones, S. L. (1992), 'Implementing lazy functional programming languages on stock hardware: the spineless tagless G-machine', *Journ. Funct. Progr.* **2**(2), 127–202.

Plotkin, G. D. (1975), 'Call-by-name, call-by-value, and the λ-calculus', *Theor. Comp. Sci.* **1**, 125–159.

Plotkin, G. D. (1977), 'LCF considered as a programming language', *Theor. Comp. Sci.* **5**, 223–255.

Rose, K. H. (1996), Operational Reduction Models for Functional Programming Languages, PhD thesis, DIKU, Dept. of Computer Science, Univ. of Copenhagen, Universitetsparken 1, DK-2100 København Ø. DIKU report 96/1.

Rosen, B. K. (1973), 'Tree-manipulating systems and Church-Rosser theorems', *Journ. ACM* **20**(1), 160–187.

Schönfinkel, M. (1924), 'Über die Bausteine der mathematischen Logik', *Math. Ann.* **92**, 305–316.

Sestoft, P. (1997), 'Deriving a lazy abstract machine', *Journ. Funct. Progr.* **7**(3).
⟨URL:*ftp://ftp.dina.kvl.dk/pub/Staff/Peter.Sestoft/papers/amlazy5.ps.gz*⟩

Turing, A. M. (1936), On computable numbers, with an application to the entscheidungsproblem, *in* 'Proc. London Math. Soc.', Vol. 42 of *2*, pp. 230–265.

Turner, D. A. (1979), 'A new implementation technique for applicative languages', *Software and Practice and Experience* **9**, 31–49.

Wadsworth, C. (1971), Semantics and pragmatics of the lambda calculus, PhD thesis, Oxford.

Recent Publications in the BRICS Report Series

RS-96-56 **Zine-El-Abidine Benaissa, Pierre Lescanne, and Kristoffer H. Rose.** ***Modeling Sharing and Recursion for Weak Reduction Strategies using Explicit Substitution*****. December 1996. 35 pp. Appears in Kuchen and Swierstra, editors,** ***8th International Symposium on Programming Languages, Implementations, Logics, and Programs*****, PLILP '96 Proceedings, LNCS 1140, 1996, pages 393–407.**

RS-96-55 **Kåre J. Kristoffersen, François Laroussinie, Kim G. Larsen, Paul Pettersson, and Wang Yi.** ***A Compositional Proof of a Real-Time Mutual Exclusion Protocol*****. December 1996. 14 pp. To appear in Dauchet and Bidoit, editors,** ***Theory and Practice of Software Development. 7th International Joint Conference CAAP/FASE*****, TAPSOFT '97 Proceedings, LNCS, 1997.**

RS-96-54 **Igor Walukiewicz.** ***Pushdown Processes: Games and Model Checking*****. December 1996. 31 pp. Appears in Alur and Henzinger, editors,** ***8th International Conference on Computer-Aided Verification*****, CAV '96 Proceedings, LNCS 1102, 1996, pages 62–74.**

RS-96-53 **Peter D. Mosses.** ***Theory and Practice of Action Semantics*****. December 1996. 26 pp. Appears in Penczek and Szalas, editors,** ***Mathematical Foundations of Computer Science: 21st International Symposium*****, MFCS '96 Proceedings, LNCS 1113, 1996, pages 37–61.**

RS-96-52 **Claus Hintermeier, Hélène Kirchner, and Peter D. Mosses.** ***Combining Algebraic and Set-Theoretic Specifications (Extended Version)*****. December 1996. 26 pp. Appears in Haveraaen, Owe and Dahl, editors,** ***Recent Trends in Data Type Specification: 11th Workshop on Specification of Abstract Data Types, joint with 8th COMPASS Workshop*****, Selected Papers, LNCS 1130, 1996, pages 255–274.**

RS-96-51 **Claus Hintermeier, Hélène Kirchner, and Peter D. Mosses.** ***R^n- and G^n-Logics*****. December 1996. 19 pp. Appears in Gilles, Heering, Meinke and Möller, editors,** ***Higher-Order Algebra, Logic, and Term-Rewriting: 2nd International Workshop*****, HOA '95 Proceedings, LNCS 1074, 1996, pages 90–108.**