BRICS

Basic Research in Computer Science

BRICS LS-96-3 K. H. Rose: Explicit Substitution – *Tutorial & Survey*

Explicit Substitution – *Tutorial & Survey*

Kristoffer H. Rose

BRICS Lecture Series **LS-96-3**

ISSN 1395-2048 **October 1996**

See back inner page for a list of recent publications in the BRICS Lecture 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 the World Wide Web and anonymous FTP:

`http://www.brics.dk/`
`ftp://ftp.brics.dk/pub/BRICS`
This document in subdirectory `LS/96/3/`

Typeset by the author with the TEX typesetting system (Knuth 1984), using the LATEX 2ε format (Lamport 1994), with several essential packages, notably AMS-LATEX (American Mathematical Society 1995) and XY-pic (Rose and Moore 1995).

Explicit Substitution – Tutorial & Survey

Kristoffer H. Rose

Kristoffer Høgsbro Rose

⟨krisrose@brics.dk⟩
W^3 ⟨URL: http://www.brics.dk/~krisrose/⟩

BRICS[1]
Dept. of Computer Science
University of Aarhus
Ny Munkegade, building 540
8000 Århus C, Denmark

[1] Basic Research in Computer Science,
Centre of the Danish National Research Foundation,

Preface

Gentle Reader, the present notes constitute the material of the mini-course on explicit substitution on October 27, 1996, at BRICS in the Department of Computer Science at the University of Aarhus. I hope that they will prove useful in other contexts as well.

Goals. The purpose of the mini-course (as well as these notes) is to give a coherent overview of the area of explicit substitution and some applications. We will focus on the *operational* or *syntactic* side of things, in particular we will not cover the areas of semantics and type systems for explicit substitution calculi. Emphasis is put on providing a universal understanding of the very different techniques used by various authors in the area.

Overview. These notes have five chapters that I present as five lectures:

Plan for Sunday, October 27, 1996		
Time	Lecture	Chapter
10–11	Explicit Substitution Rules	1
11–12	Explicit Binding	2
12–13	Explicit Addresses	3
15–16	Generalising to Higher-Order Rewriting	4
16–17	Strategies and Abstract Machines	5

This is intended as an advanced course where people already know some rewriting, inductive notations, and the λ-calculus, however, the necessary preliminaries are summarised in the appendix.

Chapter 1 is fundamental and is assumed known in the following chapters: It explains of explicit substitution relative to the usual untyped λ-calculus, *i.e.*, how usual substitution (using variable names to express λ-binding) can be understood as a "syntactic" explicit concept rather than a "semantic" meta-notion.

The next three chapters are largely independent: Chapter 2 extends explicit substitution to also understand *binding* syntactically, which makes it possible to relate to most other published calculi of explicit substitution. In Chapter 3 we go even further and make *memory* explicit by allowing addresses in terms. In Chapter 4 we show how these notions generalise to *higher-order rewriting* in the form of combinatory reduction systems.

Finally, in Chapter 5, we put everything together to show the difficult part of how one can define the notion of a strategy and derive an abstract machine for functional programming languages with explicit binding and recursion.

Each chapter is concluded with a discussion, and in the bibliography the most central entries have been annotated with a short commentary. Last there is a comprehensive index including people, definitions, and notations.

Credit. The notes are (slightly revised) sections from Rose (1996b) on explicit substitution, combined with material from several publications on the subject where the author has been involved (Bloo and Rose 1996, Benaissa, Lescanne and Rose 1996, Rose 1996a). This is a manifestation of the fact that this course would not have been without the benefit of interaction with the very lively and varied community involved in explicit substitution. I foremost wish to thank my coauthors, Zine-El-Abidine Benaissa, Roel Bloo, and Pierre Lescanne, for our pleasant collaborations, the result of which is included in several sections. Finally, I am grateful for the environment provided by BRICS where courses such as this are not only possible but worthwhile!

Kristoffer Høgsbro Rose
Århus, September 1996

Contents

List of Figures

Chapter 1

Explicit Substitution Rules

The λ-calculus (Church 1941, Barendregt 1984) is a convenient model of the computable functions as well as a compact notation for algorithms. But it does not provide any means for observing *operational* properties of the execution of these algorithms such as the time or space consumption. The reason for this is that substitution operation implicit in β-contraction is not *primitive*: it can neither be implemented in constantly bounded time nor space, since the size of the abstraction body in which substitutions happen is not bounded.

Curry and Feys (1958) proposed a solution to this, namely finding a syntactic model where the reduction steps are primitive in this sense, and proposed *Combinatory Logic*. However, in CL the number of primitive steps can be vastly larger than the actual work required by λ-reducers. The first realistic syntactic model of λ-reduction was Curien's (1986) *Categorical Combinatory Logic* which formed the basis of the now seminal work of Abadi, Cardelli, Curien and Lévy (1991) on $\lambda\sigma$, the first *explicit substitution calculus*. With these calculi it is possible to reason theoretically about operational aspects of evaluation in functional and other high-level programming languages.

This chapter surveys "naïve explicit substitution" for the λ-calculus, which is nevertheless sufficiently expressive that most considerations discussed in the literature can be formulated from it, and indeed we will explain most published results on explicit substitution in the context of naïve substitution.

Plan. We start in Section 1.1 by refining the λ-calculus to λxgc, a calculus of explicit substitution; considerable care is taken to show that the explicit substitution reduction relation is a *conservative extension* of the λ-calculus and thus inherits most of its properties, notably confluence and (weak) normalisation (see

Appendix A.1 for an introduction to the abstract reduction concepts used). A particular property of λxgc is the rule for *explicit garbage collection* which will turn out to play an important rôle in the following Section 1.2 where we show that the set of strongly normalising λ-terms is also λxgc-strongly normalising, or *preserves strong normalisation* (PSN), a property that is surprisingly difficult to establish; we also explain a simplified version of the counterexample to PSN for $\lambda\sigma$ found by Melliès (1995). Section 1.3 discusses the usefulness of the obtained results and list some open problems and interesting further directions.

The chapter covers the results reported in Bloo and Rose (1995).

1.1 Explicit Substitution

This section presents the λxgc-calculus: an *explicit substitution calculus* in the tradition of $\lambda\sigma$ of Abadi et al. (1991) but retaining variable *names* instead of using indices à la de Bruijn (1972). This makes the calculus simpler by making it possible to use only 'naïve direct substitution' following Rose (1992). Furthermore, λxgc shares with λs of Kamareddine and Ríos (1995) and $\lambda\upsilon$ of Lescanne (1994) that there is no syntax for explicit composition of substitutions (we elaborate on this in section 1.2). Finally, the calculus has *explicit garbage collection* (of Rose 1992) since it is useful and fairly easy to specify using names but mostly because it makes the proof of preservation of strong normalisation simpler and direct. We show, however, that garbage collection is not observable in any way.

We first present the λx-terminology which we will use throughout the chapter, and generalise some standard related concepts from the λ-calculus of section A.3 to them. Then we present λxgc-reduction. Finally we analyse properties of the substitution and garbage collection subrelations and explain the relation to standard substitution and β-reduction, showing that the reduction relation of λxgc is a conservative extension of $\xrightarrow[\beta]{}$, from which confluence of λxgc follows.

1.1.1 Definition (λx-preterms). The λx-preterms is the extension of the λ-preterms (ranged over by $MNPQR$) defined inductively by

$$M \ ::= \ x \ \mid \ \lambda x.M \ \mid \ MN \ \mid \ M\langle x := N\rangle$$

where $xyzvw$ 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 $\lambda x.((MN)(P\langle y := Q\rangle))$ (thus xs in all of the subterms M, N, P, and Q, are bound by the λ).

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 $\xrightarrow[\beta]{}$ (as defined on the pure terms only, of course, *cf.* Definition A.3.6). All the usual standard concepts generalise naturally.

1.1.2 Definitions (λx-terms).

1. The *free variable* set of a λx-preterm M is denoted $\mathrm{fv}(M)$ and defined inductively over M by (the usual) $\mathrm{fv}(x) = \{x\}$, $\mathrm{fv}(\lambda x.M) = \mathrm{fv}(M)\setminus\{x\}$, $\mathrm{fv}(MN) = \mathrm{fv}(M) \cup \mathrm{fv}(N)$, and (the new)
$$\mathrm{fv}(M\langle x := N\rangle) = (\mathrm{fv}(M)\setminus\{x\}) \cup \mathrm{fv}(N)\ .$$
A λx-preterm M is *closed* iff $\mathrm{fv}(M) = \varnothing$.

2. The result of *renaming* all free occurrences of y in M to z is written $M[y := z]$ and defined inductively over M and defined inductively over M by (the usual) $x[y := z] = z$ if $x = y$, $x[y := z] = x$ if $x \neq y$, $(\lambda x.M)[y := z] = \lambda x'.M[x := x'][y := z]$ with $x' \notin \mathrm{fv}(\lambda x.M) \cup \{y, z\}$, $(MN)[y := z] = (M[y := z])\,(N[y := z])$, and (the new)
$$(M\langle x := N\rangle)[y := z] = M[x := x'][y := z]\ \langle x' := N[y := z]\rangle$$
with $x' \notin \mathrm{fv}(\lambda x.M) \cup \{y, z\}$.

3. That two terms are *α-equivalent* is written $M \equiv N$. This means that they are identical except for renaming of bound variables, defined inductively by (the usual) $x \equiv x$, $\lambda x.M \equiv \lambda y.N$ if $M[x := z] \equiv N[y := z]$ for $z \notin \mathrm{fv}(MN)$, $MN \equiv PQ$ if $M \equiv P$ and $N \equiv Q$, and (the new)
$$M\langle x := N\rangle \equiv P\langle y := Q\rangle \quad \text{if } N \equiv Q \text{ and } M[x := z] \equiv P[y := z] \text{ for } z \notin \mathrm{fv}(MP).$$

4. The set of λx-*terms*, Λx, is the set of λx-preterms modulo $\equiv$. The set of *closed* λx-*terms*, Λx°, is the subset of Λ where the representatives are closed preterms.

1.1.3 Remark. One might argue that

$$\mathrm{fv}(M\langle x := N\rangle) = \begin{cases} \mathrm{fv}(M) & \text{if } x \notin \mathrm{fv}(M), \\ (\mathrm{fv}(M)\backslash\{x\}) \cup \mathrm{fv}(N) & \text{if } x \in \mathrm{fv}(M) \end{cases} \tag{$*$}$$

is a better definition since when $x \notin \mathrm{fv}(M)$ it is impossible for a substitution $\langle y := P\rangle$ in $M\langle x := N\rangle\langle y := P\rangle$ to substitute P for any variable in N. We will not do this, however, since it would interfere with garbage collection in an unnatural way, for example it would allow the reduction $x\langle y := z\rangle\langle z := v\rangle\langle z := w\rangle \rightarrow x\langle y := z\rangle\langle z := w\rangle$ which seems to change the binding of the inner z.

Also note that $M\langle x := N\rangle$ has the same free variables as $(\lambda x.M)N$; in these terms free occurrences of x in M are bound by $\_\langle x := N\rangle$ respectively $\lambda x.\_$. Using $(*)$ would mean that contraction of a redex could remove free variables from a term which seems undesirable.

As usual we will use the variable Convention A.3.4.

1.1.4 Definitions (λxgc-reduction). Define the following reductions on Λx.

1. $\xrightarrow[b]{}$, *substitution generation*, is the contextual closure (modulo $\equiv$) of

$$(\lambda x.M)N \rightarrow M\langle x := N\rangle \tag{b}$$

2. $\xrightarrow[x]{}$, *explicit substitution*, is defined as the contextual closure (modulo $\equiv$) of the union of

$$\begin{aligned} x\langle x := N\rangle &\rightarrow N && && \text{(xv)} \\ x\langle y := N\rangle &\rightarrow x && \text{if } x \not\equiv y && \text{(xvgc)} \\ (\lambda x.M)\langle y := N\rangle &\rightarrow \lambda x.M\langle y := N\rangle && && \text{(xab)} \\ (M_1M_2)\langle y := N\rangle &\rightarrow M_1\langle y := N\rangle\ M_2\langle y := N\rangle && && \text{(xap)} \end{aligned}$$

3. $\xrightarrow[gc]{}$, *garbage collection*, is the contextual closure (modulo $\equiv$) of

$$M\langle x := N\rangle \rightarrow M \qquad \text{if } x \notin \mathrm{fv}(M) \tag{gc}$$

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

4. λxgc-*reduction* is $\xrightarrow[bxgc]{} = \xrightarrow[b]{} \cup \xrightarrow[x]{} \cup \xrightarrow[gc]{}$, λx-*reduction* is $\xrightarrow[bx]{} = \xrightarrow[b]{} \cup \xrightarrow[x]{}$, and $\xrightarrow[xgc]{} = \xrightarrow[x]{} \cup \xrightarrow[gc]{}$.

The following is immediate from the definition.

1.1.5 Propositions. 1. If $M, N \in \Lambda$, $M\langle x := N\rangle \twoheadrightarrow_{x} M[x := N]$. 2. $P \twoheadrightarrow_{x} Q$ implies $Q \in \Lambda$.

This in combination with the fact that substitution normal forms are unique (more elaborate proofs will be given below) motivates the following.

1.1.6 Notation (normal forms). Since $\xrightarrow[x]{}$UN we will use the notation $\downarrow_x(M)$ for the $\xrightarrow[x]{}$-nf of M and where we say M is *pure* if $M \equiv \downarrow_x(M) \in \Lambda$ (or equivalently $M \twoheadrightarrow_{x} M$). Similarly $\downarrow_{gc}(M)$ is the $\xrightarrow[gc]{}$-nf of M and we say M is *garbage-free* if $M \equiv \downarrow_{gc}(M)$.

Properties of λxgc-reduction and $\xrightarrow[xgc]{}$ will imply the same for λx-reduction and $\xrightarrow[x]{}$ since omitting (gc) rarely makes a difference (we explain why it is important below). We only discuss the more general λxgc-reduction in this section.

1.1.7 Remark (implicit renaming). The above reduction definition is based directly on the usual definition of β-reduction with substitution, except that it is made explicit by reducing the implicit definition of 'meta-substitution' to an explicit syntactic substitution without changing anything else. In particular implicit renaming of variables is silently assumed following Convention A.3.4. In addition we have just added the definition of garbage collection directly (introducing additional overlap between the rules). The usual definition of substitution (*cf.* Barendregt 1984, 2.1.15) is very carefully worded in order to avoid problems with variable *capture* and *clash*, and this is reflected in our definition as well: in rule (xab) the variable convention avoids variable clash (by requiring that $x \not\equiv y$) and it avoids capture by requiring that $x \notin \mathrm{fv}(N)$. Alternatively, this could be achieved by *explicit renaming*, *i.e.*, by using the following equivalent but more cumbersome formulation of the rule:

$$(\lambda x.M)\langle y := N\rangle \to \lambda x'.M[x := x']\langle y := N\rangle \quad x' \notin \mathrm{fv}(N) \cup \mathrm{fv}(\lambda x.M) \cup \{y\}$$

where the required renaming is made explicit, which creates the need for the shown constraints as follows:

- If there are free x in N then just moving the explicit substitution inside $\lambda x \ldots$ would capture them. This is prevented by renaming x in M to x', requiring that $x' \notin \mathrm{fv}(N)$.
- However, we must also ensure that the renaming itself does not clash with an existing free variable in M as this would then be captured by the created $\lambda x' \ldots$. This is ensured by the constraint $x' \notin \mathrm{fv}(\lambda x.M)$.

- Finally, the new variable should not clash with y, hence the requirement $x' \notin \{y\}$.

Notice that if $x \notin \mathrm{fv}(N)$ and $x \not\equiv y$, there is no need to rename: $x \equiv x'$ is acceptable.

One could attempt to make renaming explicit by changing the rule to

$$(\lambda x.M)\langle y := N\rangle \to \lambda x'.M\langle x := x'\rangle\langle y := N\rangle \quad x' \notin \mathrm{fv}(N) \cup \mathrm{fv}(\lambda x.M) \cup \{y\}$$

but this is not much better as the problem of allocating 'fresh' variables remains.

1.1.8 Example. To give the reader an idea of this new reduction relation we present the full λxgc-reduction graph of the term $(\lambda y.(\lambda z.z)y)x$ in Figure 1.1 (the pure terms are boxed). For comparison the full β-reduction graph of this term is

$$\begin{array}{c} (\lambda y.(\lambda z.z)y)x \\ {\scriptstyle\beta}\downarrow\;\downarrow{\scriptstyle\beta} \\ (\lambda z.z)x \\ {\scriptstyle\beta}\downarrow \\ x \end{array}$$

The length of each path in the reduction graph is a measure for the complexity of the corresponding evaluation. Notice that some paths do not pass by the intermediate term $(\lambda z.z)x$.

1.1.9 Comparison (earlier naming calculi). Two earlier expositions of explicit substitution using variable names are known to the author. The first was in the "axioms for the theory of λ-conversion" of Revesz (1985):

$$(\lambda x.x)Q \to Q \tag{β1}$$
$$(\lambda x.x)y \to x \quad \text{if } x \neq y \tag{β2}$$
$$(\lambda x.\lambda x.P)Q \to \lambda x.P \tag{β3}$$
$$(\lambda x.\lambda y.P)Q \to \lambda y.(\lambda x.P)Q \quad \text{if } x \neq y \text{ and } x \notin \mathrm{fv}(P) \text{ and } x \notin \mathrm{fv}(Q) \tag{β4}$$
$$(\lambda x.P_1P_2)Q \to (\lambda x.P_1)Q\,((\lambda x.P_2)Q) \tag{β5}$$

This relation, call it $\xrightarrow[R]{}$, is related to λx by the equation $\xrightarrow[R]{} = \xrightarrow[b]{} \cdot \xrightarrow[x]{} \cdot \twoheadleftarrow_{b}$ (the last arrow is b-expansion to normal form). The problem with Revesz's calculus for our purposes is that it does not make sense to talk about the 'substitution

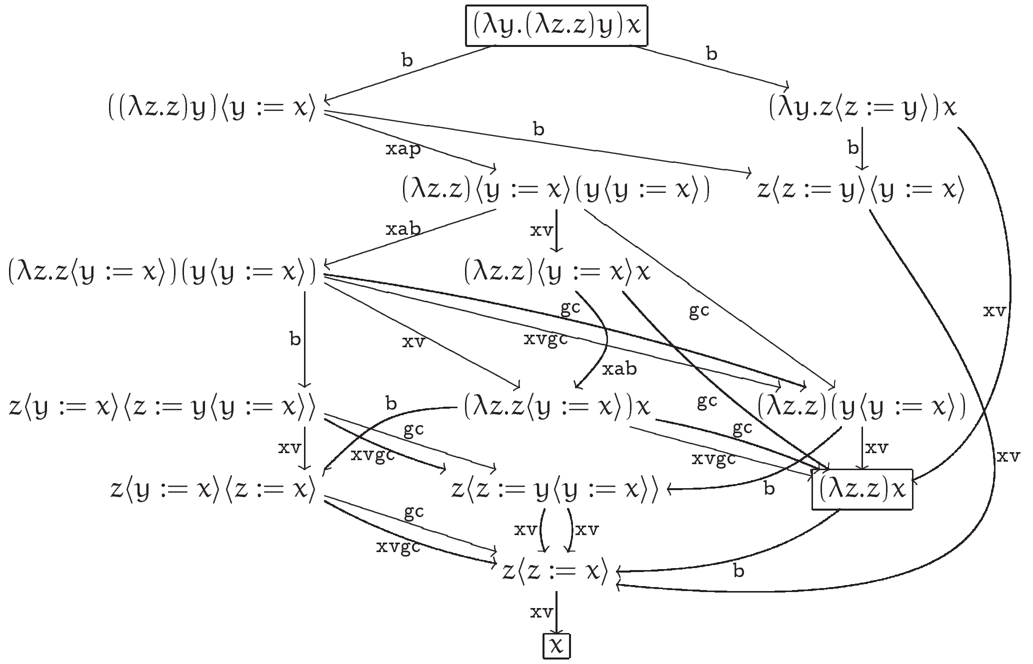


Figure 1.1: λxgc-reduction graph for $(\lambda y.(\lambda z.z)y)x$.

normal form': one cannot observe which substitutions are 'in progress'. This is essential in the treatment below.

Another calculus using names is the variant of $\lambda\sigma$ discussed briefly in section 3.3 of Abadi et al. (1991). It has the following rules:

$$
\begin{aligned}
(\lambda x.a)b &\to a[(b/x)\cdot \mathsf{id}] && \text{(Beta)}\\
x[(a/x)\cdot s] &\to a && \text{(Var1)}\\
x[(a/y)\cdot s] &\to x[s] && \text{(Var2)}\\
x[\mathsf{id}] &\to x && \text{(Var3)}\\
(ba)[s] &\to (b[s])(a[s]) && \text{(App)}\\
(\lambda x.a)[s] &\to \lambda y.(a[(y/x)\cdot s]) \quad \text{if } y \text{ does not occur in } a && \text{(Abs)}
\end{aligned}
$$

The rules correpond closely to those of λx except for the crucial difference that (Abs) introduces *explicit renaming* as discussed in Remark 1.1.7 – the 'names' are *not* variable names in the λ-calculus sense but 'strings' with all the associated problems of allocation, *etc*, and hence difficult to describe – a conclusion also reached by Abadi et al.

Next we establish 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 and traditional (implicit) substitution, and garbage collection.

1.1.10 Propositions. 1. $\xrightarrow[b]{}$SN, 2. $\xrightarrow[b]{}\Diamond$, 3. $\xrightarrow[xgc]{}$SN, 4. $\xrightarrow[xgc]{}$CR, and 5. $\xrightarrow[xgc]{}$UN.

Proof. $\xrightarrow[b]{}\Diamond$ is easy since $\xrightarrow[b]{}$ is orthogonal and linear, $\xrightarrow[b]{}$SN by observing that each $\xrightarrow[b]{}$-reduction decreases the number of b-redexes in a term.

$\xrightarrow[xgc]{}$SN is shown by finding a map $h : \Lambda x \to \mathbb{N}$ such that for all $M \xrightarrow[xgc]{} N$: $h(M) > h(N)$. 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 $h(M\langle x := N\rangle) = h(M) \times (h(N) + 1)$.

$\xrightarrow[xgc]{}$LC follows from the observation that the three critical pairs are resolved like this:

$$x\langle y := N\rangle \underset{gc}{\overset{xvgc}{\rightrightarrows}} x$$

where $y \not\equiv x$,

$$\begin{tikzcd}
 & \lambda x.M\langle y := N\rangle \arrow[dr, "\mathrm{gc}"] & \\
(\lambda x.M)\langle y := N\rangle \arrow[ur, "\mathrm{xab}"] \arrow[rr, "\mathrm{gc}"'] & & \lambda x.M
\end{tikzcd}$$

where $y \notin \mathrm{fv}(\lambda x.M)$, and

$$\begin{tikzcd}
 & M_1\langle y := N\rangle\ M_2\langle y := N\rangle \arrow[dr, two heads, "\mathrm{gc}"] & \\
(M_1M_2)\langle y := N\rangle \arrow[ur, "\mathrm{xap}"] \arrow[rr, "\mathrm{gc}"'] & & (M_1M_2)
\end{tikzcd}$$

where $y \notin \mathrm{fv}(M_1M_2)$. SN and LC imply CR and UN by Lemma A.2.9. $\square$

Now we will 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 $P[x := Q]$.

1.1.11 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. In particular $\downarrow_x(M\langle x := N\rangle) \equiv M[x := N]$ for pure M and N.

Proof. We show by induction on the number of symbols in $M, N_1, \ldots, N_n$ that

$$\downarrow_x(M\langle x_1 := N_1\rangle \cdots \langle x_n := N_n\rangle) \equiv \downarrow_x(M)[x_1 := \downarrow_x(N_1)] \cdots [x_n := \downarrow_x(N_n)]$$

Using this as the IH we distinguish cases according to the structure of M:

Case $M \equiv x$, $n = 0$: $\downarrow_x(x) \equiv \downarrow_x(x)$.

Case $M \equiv x$, $n \geq 1$: If $x \not\equiv x_1$ then

$$\begin{aligned}
\downarrow_x(x\langle x_1 := N_1\rangle \cdots \langle x_n := N_n\rangle) &\equiv \downarrow_x(x\langle x_2 := N_2\rangle \cdots \langle x_n := N_n\rangle) \\
&\stackrel{\mathrm{IH}}{\equiv} \downarrow_x(x)[x_2 := \downarrow_x(N_2)] \cdots [x_n := \downarrow_x(N_n)] \\
&\equiv \downarrow_x(x)[x_1 := \downarrow_x(N_1)] \cdots [x_n := \downarrow_x(N_n)]\ ;
\end{aligned}$$

if $x \equiv x_1$ then

$$\begin{aligned}
\downarrow_x(x\langle x_1 := N_1\rangle \cdots \langle x_n := N_n\rangle) &\equiv \downarrow_x(N_1\langle x_2 := N_2\rangle \cdots \langle x_n := N_n\rangle) \\
&\stackrel{\text{IH}}{\equiv} \downarrow_x(N_1)[x_2 := \downarrow_x(N_2)] \cdots [x_n := \downarrow_x(N_n)] \\
&\equiv x[x := \downarrow_x(N_1)] \cdots [x_n := \downarrow_x(N_n)] \\
&\equiv \downarrow_x(x)[x_1 := \downarrow_x(N_1)] \cdots [x_n := \downarrow_x(N_n)]
\end{aligned}$$

Cases $M \equiv PQ$ and $M \equiv \lambda x.P$: Easy.

Case $M \equiv P\langle y := Q\rangle$: The number of symbols in $P\langle y := Q\rangle, N_1, \dots, N_n$ is bigger than the number of symbols in P, Q, $N_1, \dots, N_n$ and (for the second use of IH) bigger than the number of symbols in P, Q. Hence

$$\begin{aligned}
&\downarrow_x(P\langle y := Q\rangle\langle x_1 := N_1\rangle \cdots \langle x_n := N_n\rangle) \\
&\quad\stackrel{\text{IH}}{\equiv} \downarrow_x(P)[y := \downarrow_x(Q)][x_1 := \downarrow_x(N_1)] \cdots [x_n := \downarrow_x(N_n)] \\
&\quad\stackrel{\text{IH}}{\equiv} \downarrow_x(P\langle y := Q\rangle)[x_1 := \downarrow_x(N_1)] \cdots [x_n := \downarrow_x(N_n)]
\end{aligned}$$

□

An interesting consequence of representation is the following:

1.1.12 Corollary (substitution lemma).

$$M\langle x := N\rangle\langle y := P\rangle \underset{\text{xgc}}{\twoheadleftarrow\!\!\twoheadrightarrow} M\langle y := P\rangle\langle x := N\langle y := P\rangle\rangle$$

Proof. Follows from Lemma 1.1.11 by the λ-calculus substitution lemma. □

For the pure calculus this conversion degenerates to an identity which is the usual pure λ-calculus substitution lemma.

Next several useful results, including the relationship of $\underset{x}{\rightarrow}$ to garbage collection.

1.1.13 Propositions. For any M,

1. If $M \underset{\text{gc}}{\rightarrow} N$ then $\text{fv}(M) \supseteq \text{fv}(N)$ and $\downarrow_x(M) \equiv \downarrow_x(N)$.
2. If $M \underset{x}{\rightarrow} N$ then $\text{fv}(M) \supseteq \text{fv}(N)$.
3. If $M \underset{b}{\rightarrow} N$ then $\text{fv}(M) = \text{fv}(N)$.
4. If M is garbage-free then $\text{fv}(M) = \text{fv}(\downarrow_x(M))$.

Proofs. The two free variable inclusions and 3 follow by an easy induction on M. The preservation of $\downarrow_x$ is shown by induction on the structure of M; the crucial case is $P\langle x := Q\rangle \xrightarrow[gc]{} P$ where $x \notin fv(P)$: then by Lemma 1.1.11 $\downarrow_x(P\langle x := Q\rangle) \equiv \downarrow_x(P)[x := \downarrow_x(Q)] \equiv \downarrow_x(P)$ since $x \notin fv(P) \supseteq fv(\downarrow_x(P))$.

Concerning 4, we first note that there is a reduction path

$$M \equiv M'_0 \xrightarrow[x]{} M_1 \xrightarrow[gc]{}\!\!\!\!\rightarrow M'_1 \xrightarrow[x]{} M_2 \xrightarrow[gc]{}\!\!\!\!\rightarrow M'_2 \xrightarrow[x]{} \cdots \xrightarrow[gc]{}\!\!\!\!\rightarrow M'_n \equiv \downarrow_x(M)$$

where $M_{i+1} \equiv M'_{i+1}$ if $M'_i \xrightarrow[x]{} M_{i+1}$ is not an (xab) step, and $M_{i+1} \overset{=}{\xrightarrow[gc]{}} M'_{i+1}$ if $M'_i \xrightarrow[x]{} M_{i+1}$ is a (xab) step, and all M'_i are garbage-free. For instance if $M'_i \equiv (PQ)\langle x := R\rangle$, $M_{i+1} \equiv (P\langle x := R\rangle)(Q\langle x := R\rangle)$, $x \in fv(P)$, $x \notin fv(Q)$, then $M_{i+1} \xrightarrow[gc]{} M'_{i+1} \equiv (P\langle x := R\rangle)Q$ and M'_{i+1} is garbage-free iff M'_i is garbage-free. Now it is easy to show by induction on the structure of M'_i, that M'_i garbage-free implies M'_{i+1} garbage-free and $fv(M'_i) = fv(M'_{i+1})$. $\square$

That the two $\supseteq$ cannot be strenghtened to $=$ can see by considering the reduction $x\langle y := z\rangle \longrightarrow x$ valid for both $\xrightarrow[x]{}$ and $\xrightarrow[gc]{}$.

A consequence of Proposition 1.1.13.1 is that we do not need to introduce a special notation for $\xrightarrow[xgc]{}$-normal forms since any pure term is also garbage-free, *i.e.*, $\xrightarrow[xgc]{}\!\!\!\!\twoheadrightarrow\!\!\!\!| = \xrightarrow[x]{}\!\!\!\!\twoheadrightarrow\!\!\!\!|$.

1.1.14 Remark (on garbage collection). Considering the above, one might ask why $\xrightarrow[gc]{}$ is needed at all when its effect seems obtainable by $\xrightarrow[x]{}$? The answer is that its effect is not obtainable by $\xrightarrow[x]{}$: intuitively $\xrightarrow[gc]{}$ does a single global garbage collection whereas $\xrightarrow[x]{}$ moves a substitution into a term, working a bit of the way towards garbage collection. Formally, not only is $\xrightarrow[x]{} \not\subseteq \xrightarrow[gc]{}\!\!\!\!\rightarrow$ but also $\xrightarrow[gc]{} \not\subseteq \xrightarrow[x]{}\!\!\!\!\rightarrow$: the second inclusion fails because a single $\xrightarrow[gc]{}$ can remove a single substitution which can't be removed by $\xrightarrow[x]{}\!\!\!\!\rightarrow$ when it is not the innermost, *e.g.*, $x\langle x := y\rangle\langle v := w\rangle \xrightarrow[gc]{} x\langle x := y\rangle$ which can't be done by $\xrightarrow[x]{}\!\!\!\!\rightarrow$ (also see Remark 1.1.3). This is witnessed in Figure 1.1 where most but not all $\xrightarrow[gc]{}$-arrows are adjacent to an $\xrightarrow[xvgc]{}$-arrow.

Furthermore, $\xrightarrow[gc]{}$ permits more efficient reduction through the use of clever strategies. One of the problems with explicit substitution is that there is a tendency to create substitutions that do not bind a variable, as is the case in $(x(fyyy))\langle x := I\rangle \xrightarrow[x]{} x\langle x := I\rangle\ (fyyy)\langle x := I\rangle$ Without $\xrightarrow[gc]{}$, the reduction to normal form of the latter term will involve distributing $\langle x := I\rangle$ three times over $fyyy$ and then deleting it four times using $\xrightarrow[xvgc]{}$. Using $\xrightarrow[gc]{}$ make it possible to throw away the useless substitution $\langle x := I\rangle$ in $(fyyy)\langle x := I\rangle$ right after its creation, and hence $fyyy$ can be reached more efficiently with respect to the length of the reduction path, as well as with respect to the size of the terms along the reduction path.

Finally, garbage collection makes a modular proof of preservation of strong normalisation possible in the next section, something it is not clear whether one could do without.

Now that we have established the properties of the substitution subrelation, we can compare to the ordinary (pure) β-reduction. We show that $\xrightarrow[\mathrm{bxgc}]{}$ is a conservative extension of $\xrightarrow[\beta]{}$, and thus confluent. We first relate single reduction steps and finally full reduction.

1.1.15 Proposition. For pure M, 1. $\begin{tikzcd} M \arrow[r, "b"'] \arrow[dr, dotted, bend right, "\beta"'] & \blacksquare \arrow[d, dotted, twoheadrightarrow, "x"] \\ & \square \end{tikzcd}$, and 2. $\begin{tikzcd} M \arrow[rr, "\beta"] \arrow[dr, dotted, "b"'] & & \blacksquare \\ & \square \arrow[ur, dotted, twoheadrightarrow, "x"'] & \end{tikzcd}$.

Proof. Both follow from the observation that we can choose to contract the same β-redex with $\xrightarrow[\beta]{}$ and $\xrightarrow[b]{}$, and that the result of the $\xrightarrow[b]{}$-reduction will have exactly one substitution in it: we then use $\xrightarrow[x]{}$UN and Lemma 1.1.11. □

Next Proposition 1.1.15.1 is generalised to non-pure terms because we will need this in the confluence proof below. In fact we also strenghten it slightly in the special case where the contracted term is garbage-free since this will be needed for the proof of preservation of strong normalisation in the following section.

1.1.16 Lemmas (projection). 1. For all λx-terms M, $\begin{tikzcd} M \arrow[r, "b"] \arrow[d, twoheadrightarrow, "x"] & N \arrow[d, twoheadrightarrow, "x"'] \\ \downarrow_x(M) \arrow[r, dotted, twoheadrightarrow, "\beta"'] & \downarrow_x(N) \end{tikzcd}$

2. For garbage-free M, $\begin{tikzcd} M \arrow[r, "b"] \arrow[d, twoheadrightarrow, "x"] & N \arrow[d, twoheadrightarrow, "x"'] \\ \downarrow_x(M) \arrow[r, dotted, "+"' above, "\beta"'] & \downarrow_x(N) \end{tikzcd}$

Proof. We first prove 2 by induction on the structure of M:

Case $M \equiv (\lambda x.P)Q$, $N \equiv P\langle x := Q\rangle$:

$$\downarrow_x(M) \equiv (\lambda x.\downarrow_x(P))\downarrow_x(Q) \xrightarrow[\beta]{} \downarrow_x(P)[x := \downarrow_x(Q)] \stackrel{\text{Lemma 1.1.11}}{\equiv} \downarrow_x(P\langle x := Q\rangle)$$

Case $M \equiv PQ$, $N \equiv P'Q$: $\downarrow_x(M) \equiv \downarrow_x(P)\downarrow_x(Q) \overset{\text{IH}}{\xrightarrow[\beta]{+}} \downarrow_x(P')\downarrow_x(Q)$.

Case $M \equiv PQ$, $N \equiv PQ'$: Analogous.

Case $M \equiv \lambda x.P$, $N \equiv \lambda x.P'$: Analogous.

Case $M \equiv P\langle x := Q\rangle$, $N \equiv P'\langle x := Q\rangle$: By IH we know that $\downarrow_x(P) \overset{+}{\underset{\beta}{\to}} \downarrow_x(P')$, hence also $\downarrow_x(P)[x := \downarrow_x(Q)] \overset{+}{\underset{\beta}{\to}} \downarrow_x(P')[x := \downarrow_x(Q)]$. Thus $\downarrow_x(M) \overset{\text{Lemma 1.1.11}}{\equiv} \downarrow_x(P)[x := \downarrow_x(Q)] \overset{+}{\underset{\beta}{\to}} \downarrow_x(P')[x := \downarrow_x(Q)] \overset{\text{Lemma 1.1.11}}{\equiv} \downarrow_x(N)$.

Case $M \equiv P\langle x := Q\rangle$, $N \equiv P\langle x := Q'\rangle$: By IH we know that $\downarrow_x(Q) \overset{+}{\underset{\beta}{\to}} \downarrow_x(Q')$, hence also $\downarrow_x(P)[x := \downarrow_x(Q)] \overset{+}{\underset{\beta}{\to}} \downarrow_x(P)[x := \downarrow_x(Q')]$ since by the fact that M is garbage-free and Proposition 1.1.13.4, $x \in \mathrm{fv}(\downarrow_x(P))$. Thus $\downarrow_x(M) \overset{\text{Lemma 1.1.11}}{\equiv} \downarrow_x(P)[x := \downarrow_x(Q)] \overset{+}{\underset{\beta}{\to}} \downarrow_x(P)[x := \downarrow_x(Q')] \overset{\text{Lemma 1.1.11}}{\equiv} N$.

1 is analogous except that we do not know in the last case that $x \in \mathrm{fv}(\downarrow_x(P))$ and hence have to use $\underset{\beta}{\twoheadrightarrow}$, allowing identity ($\equiv$). ☐

The above suffices to show that λxgc is indeed a conservative extension of the $\lambda\beta$-calculus, with $\underset{x}{\twoheadrightarrow}$ as translation.

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

Proof. **Case** $\Longleftarrow$: Assume $M \underset{\beta}{\twoheadrightarrow} N$; the case then follows by induction on the length of the $\underset{\beta}{\to}$-reduction, using Proposition 1.1.15.2 in each step.

Case $\Longrightarrow$: Assume $M \underset{\mathrm{bxgc}}{\twoheadrightarrow} N$ and M, N pure. We will do induction on the length of the $\underset{\mathrm{bxgc}}{\twoheadrightarrow}$-reduction and prove

$$\begin{tikzcd}
M \arrow[r, "\mathrm{bxgc}"'] \arrow[d, equal] & M_1 \arrow[r, "\mathrm{bxgc}"'] \arrow[d, two heads, "x"'] & \cdots \arrow[r, "\mathrm{bxgc}"'] & M_{n-1} \arrow[r, "\mathrm{bxgc}"'] \arrow[d, two heads, "x"'] & N \arrow[d, equal] \\
M \arrow[r, dotted, two heads, "b"'] & \downarrow_x(M_1) \arrow[r, dotted, two heads, "b"'] & \cdots \arrow[r, dotted, two heads, "b"'] & \downarrow_x(M_{n-1}) \arrow[r, dotted, two heads, "b"'] & N
\end{tikzcd}$$

Each step in the top of each 'cell' is one of $\underset{gc}{\to}$, $\underset{x}{\to}$, and $\underset{b}{\to}$, thus each 'cell' is an instance of either Proposition 1.1.13.1, confluence of $\underset{x}{\to}$, or Lemma 1.1.16.1. ☐

An easy consequence of this is the following.

1.1.18 Corollary. $\underset{\mathrm{bxgc}}{\to}$CR.

Proof. By Proposition A.1.17.1 with $\underset{x}{\twoheadrightarrow}$ as projection. ☐

1.1.19 Discussion (locality). We argued in the introduction that explicit substitution would establish a *complexity faithful* method of reducing λ-terms. This is manifested in the definition in the fact that *contraction creates a bounded fragment of the contractum λx-term*. This will turn out to be crucial later. What we have *not* made local is the search for redexes and the duplication of subterms – the latter will be addressed in the sharing sections of this chapter; redex searching is made local in the form of explicit strategies in the next chapter, and this is generalised to CRS systems in chapter 4.

1.2 Preservation of Strong Normalisation

In this section we prove preservation of strong normalisation of untyped λ-terms (PSN) for λxgc. We first recall Melliès's (1995) counterexample to PSN of β-reduction for λσ of Abadi et al. (1991)[1] which we generalise slightly and from which we synthesise the usefulness of explicit (*i.e.*, syntactic) garbage collection. We then proceed with the proof of β-PSN in two stages: first we introduce our main technical tool to achieve the goal, 'garbage-free' reduction. We define it naïvely as a restriction of λxgc-reduction, and show that it is a confluent refinement of the λ-calculus, with β-PSN. Afterwards we show that the 'garbage collection overhead' does not provide for infinite reductions from which we conclude that unrestricted λxgc-calculus has PSN. Our proof is direct. Furthermore, by reasoning more carefully and avoiding infinite reduction sequences, the proof can be made constructive.

1.2.1 Remark (composition of substitution breaks PSN). Why not consider a 'stronger' calculus with some kind of reduction rule for composition of substitutions as λσ has? It seems poor that for λxgc the substitution lemma is only an extensional property of reduction whereas for λσ it is intensional: there is a rewrite rule which performs substitution composition in a single step.

The problem is that PSN is usually not preserved in the stronger calculi as was shown by Melliès (1995). In the most general case this is obvious: adding the rewrite rule

$$M\langle x := N\rangle\langle y := P\rangle \rightsquigarrow' M\langle y := P\rangle\langle x := N\langle y := P\rangle\rangle$$

breaks PSN since this rule immediately yields infinite rewrite possibilities whenever it is applicable. λσ essentially adds *parallel substitution* which in our

[1] We present λσ in Definition 2.2.24.

notation could be written $\langle\vec{x} := \vec{N}\rangle$ which intuitively means substitute N_1 for x_1, N_2 for x_2, ..., N_m for x_m simultaneously; the $\lambda\sigma$ rules *Clos* and *Map* are equivalent to the parallel substitution rule

$$M\langle\vec{x} := \vec{N}\rangle\langle\vec{y} := \vec{P}\rangle \rightsquigarrow'' M\langle\vec{x},\vec{y} := N_1\langle\vec{y} := \vec{P}\rangle, \ldots, N_m\langle\vec{y} := \vec{P}\rangle, P_1, \ldots, P_n\rangle$$

(this was shown by Kamareddine and Nederpelt 1993). In fact, even the simpler rule

$$M\langle x := N\rangle\langle y := P\rangle \rightsquigarrow M\langle x := N\langle y := P\rangle\rangle \qquad \text{if } y \notin \mathrm{fv}(M)$$

suffices for an exposition of Melliès's counterexample as shown in Figure 1.2 (the original term was the typable $\lambda x.(\lambda y.I(Iy))(Ix)$).

Considering these negative results on calculi with composition of substitutions, it seems important first to study a calculus of explicit substitutions without composition. Furthermore, note that all the reductions (but the first three) take place *inside garbage* and that it is essential that substitutions can be 'shifted' into garbage. Below we will show how this is impossible for λxgc, thus providing a sufficient condition for PSN.

The crucial insight provided by the counterexample is that *infinite reductions consist mainly of reductions inside garbage*. The need to avoid such has already been discerned by Kennaway and Sleep (1988) who chose not to create them at all, at the cost of restricting the evaluation order of nested β-redexes. Another way to avoid these is to perform garbage collection whenever any garbage exists. This is the main technical reason for having the reduction $\xrightarrow[\mathrm{gc}]{}$: it allows us to attack the question whether $\xrightarrow[\mathrm{bxgc}]{}$ has the PSN-property in a modular way.

Below we first study reductions where all garbage is removed as soon as possible. This has the advantage that no reductions take place inside garbage which will allow us to prove PSN by non-reflective projection. Using the intermediate result we can then prove PSN for the unrestricted reduction by carefully analyzing what can happen inside garbage, essentially showing that the 'garbage collection overhead' does not provide for infinite reductions. Our proof is direct. Furthermore, by reasoning more carefully and avoiding infinite reduction sequences, the proof can be made constructive.

1.2.2 Definition (garbage-free reduction). $\xrightarrow[\mathrm{bx}\downarrow\mathrm{gc}]{}$ is $(\xrightarrow[\mathrm{x}]{} \cdot \twoheadrightarrow\!\!\!\!\!\!\!\!\!\!\!\!_{\mathrm{gc}}\;)\cup(\xrightarrow[\mathrm{b}]{} \cdot \twoheadrightarrow\!\!\!\!\!\!\!\!\!\!\!\!_{\mathrm{gc}}\;)$, *i.e.*, the union of the composition of each of $\xrightarrow[\mathrm{x}]{}$ and $\xrightarrow[\mathrm{b}]{}$ with complete gc-reduction. We denote

Let a, b, y, y' be distinct variables. Define substitutions

$$S_0 \equiv \langle y := (\lambda y.a)b \rangle \ , \quad S_{n+1} \equiv \langle y := bS_n \rangle$$

and consider the following derivations (for simplicity we offend the variable convention but this is easily repaired):

$$\begin{aligned}
&(\lambda y.(\lambda y'.a)((\lambda y.a)b))\,((\lambda y.a)b) \\
&\qquad \twoheadrightarrow a\langle y' := (\lambda y.a)b\rangle\,\langle y := (\lambda y.a)b\rangle \\
&\qquad \rightsquigarrow a\langle y' := ((\lambda y.a)b)\,\langle y := (\lambda y.a)b\rangle\rangle \\
&\qquad \twoheadrightarrow a\langle y' := (\lambda y.a\langle y := (\lambda y.a)b\rangle)\,(b\langle y := (\lambda y.a)b\rangle)\rangle \\
&\qquad \equiv a\langle y' := (\lambda y.aS_0)(bS_0)\rangle \\
&\qquad \rightarrow a\langle y' := aS_0\langle y := bS_0\rangle\rangle \\
&\qquad \equiv a\langle y' := aS_0S_1\rangle \quad ,
\end{aligned}$$

$$\begin{aligned}
aS_0S_{m+1} &\equiv a\langle y := (\lambda y.a)b\rangle\,\langle y := bS_m\rangle \equiv a\langle y' := (\lambda y.a)b\rangle\,\langle y := bS_m\rangle \\
&\rightsquigarrow a\langle y' := ((\lambda y.a)b)\,\langle y := bS_m\rangle\rangle \\
&\twoheadrightarrow a\langle y' := (\lambda y.a\langle y := bS_m\rangle)\,(b\langle y := bS_m\rangle)\rangle \\
&\equiv a\langle y' := (\lambda y.aS_{m+1})\,(bS_{m+1})\rangle \\
&\rightarrow a\langle y' := aS_{m+1}\,\langle y := bS_{m+1}\rangle\rangle \\
&\equiv a\langle y := aS_{m+1}S_{m+2}\rangle \quad ,
\end{aligned}$$

$$\begin{aligned}
\text{and}\quad aS_{m+1}S_{n+1} &\equiv a\langle y' := bS_m\rangle\,\langle y := bS_n\rangle \\
&\rightsquigarrow a\langle y' := bS_m\langle y := bS_n\rangle\rangle \equiv a\langle y' := bS_mS_{n+1}\rangle
\end{aligned}$$

which combine into an infinite derivation in the following schematic way:

$$\begin{aligned}
(\lambda y.(\lambda y'.a)((\lambda y.a)b))\,((\lambda y.a)b) &\twoheadrightarrow \ldots S_0S_1 \ldots \twoheadrightarrow \ldots S_1S_2 \ldots \\
&\twoheadrightarrow \ldots S_0S_2 \ldots \twoheadrightarrow \ldots S_2S_3 \ldots \twoheadrightarrow \ldots S_1S_3 \ldots \\
&\twoheadrightarrow \ldots S_0S_3 \ldots \twoheadrightarrow \ldots S_3S_4 \ldots \twoheadrightarrow \cdots \\
&\ \ \vdots
\end{aligned}$$

Figure 1.2: Simplified version of Melliès's counterexample to PSN.

the garbage-free reduction calculus $\lambda x{\downarrow}gc$ (the symbol '$\downarrow$' should be read "composed with complete reduction to normal form by" so for example $\xrightarrow[\beta]{} = \xrightarrow[b\downarrow x]{}$ by Proposition 1.1.15).

We start by proving confluence. All the results here will depend heavily on the knowledge that each garbage-free reduction step is a single $\xrightarrow[x]{}$- or $\xrightarrow[b]{}$-step followed by $\xrightarrow[gc]{}\!\!\!\!\rightarrow$-reduction to gc-nf.

1.2.3 Lemmas.

1. For all λx-terms M,
$$\begin{tikzcd} M \arrow[r, "bxgc"'] \arrow[d, two heads, "gc"'] & N \arrow[d, two heads, "gc"] \\ \downarrow_{gc}(M) \arrow[r, dotted, "="', "bx\downarrow gc"] & \downarrow_{gc}(N) \end{tikzcd}.$$

2. For garbage-free M,
$$\begin{tikzcd} M \arrow[r, two heads, "bxgc"'] \arrow[dr, dotted, two heads, "bx\downarrow gc"'] & N \arrow[d, dotted, two heads, "gc"] \\ & \downarrow_{gc}(N) \end{tikzcd}.$$

Proof. 1 by easy induction on the structure of M. 2 then follows by

$$\begin{tikzcd} M \arrow[r, "bxgc"'] \arrow[dr, dotted, two heads, "bx\downarrow gc"'] & M_1 \arrow[r, two heads, "gc"'] \arrow[d, dotted, two heads, "gc"] & M_2 \arrow[r, "bxgc"'] \arrow[d, dotted, two heads, "gc"] \arrow[dr, phantom, "(*)"] & M_3 \arrow[r, two heads, "gc"'] \arrow[d, dotted, two heads, "gc"] & M_4 \arrow[r, "bxgc"'] \arrow[d, dotted, two heads, "gc"] \arrow[dr, phantom, "(*)"] & \cdots \arrow[r, two heads, "gc"'] & N \arrow[d, dotted, two heads, "gc"] \\ & \downarrow_{gc}(M_1) \equiv \downarrow_{gc}(M_2) \arrow[rr, dotted, "="', "bx\downarrow gc"] & & \downarrow_{gc}(M_3) \equiv \downarrow_{gc}(M_4) \arrow[rr, dotted, "="', "bx\downarrow gc"] & & \cdots \equiv & \downarrow_{gc}(N) \end{tikzcd}$$

using 1 several times. □

1.2.4 Theorem. $\xrightarrow[bx\downarrow gc]{}$CR.

Proof. We show $\xrightarrow[bx\downarrow gc]{}\!\!\!\!\rightarrow\Diamond$: it follows from $\xrightarrow[bx\downarrow gc]{} \subseteq \xrightarrow[bxgc]{}\!\!\!\!\rightarrow$, the confluence of $\xrightarrow[bxgc]{}$, and (two applications of) Lemma 1.2.3.2 by

$$\begin{tikzcd} & \blacksquare \arrow[dr, dotted, two heads, "bxgc"] \arrow[drr, dotted, two heads, bend left, "bx\downarrow gc"] & & \\ \blacksquare \arrow[ur, two heads, "bx\downarrow gc"] \arrow[dr, two heads, "bx\downarrow gc"'] & & \square \arrow[r, dotted, two heads, "gc"'] & \square \\ & \blacksquare \arrow[ur, dotted, two heads, "bxgc"'] \arrow[urr, dotted, two heads, bend right, "bx\downarrow gc"'] & & \end{tikzcd}$$

□

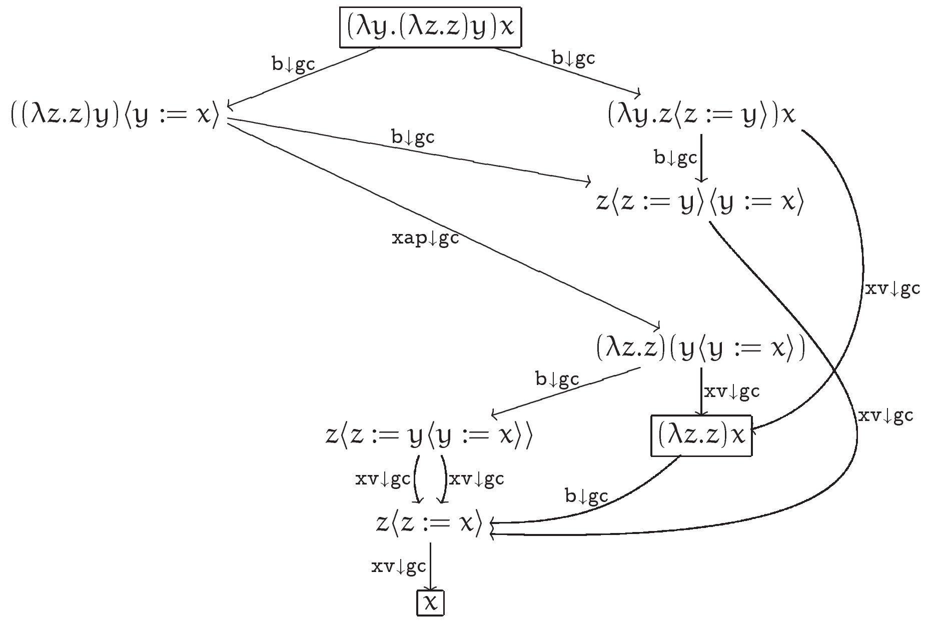


Figure 1.3: Garbage-free $\lambda x{\downarrow}gc$-reduction graph for $(\lambda y.(\lambda z.z)y)x$.

1.2.5 Example. Garbage-free reduction prohibits reduction inside garbage since any garbage has to be removed before other reductions can take place. Of course, there can be garbage collection steps inside garbage, like $v\langle w := x\langle y := z\rangle\rangle \xrightarrow[gc]{} v\langle w := x\rangle \xrightarrow[gc]{} v$. Usually, garbage-free reduction graphs are smaller than raw reduction graphs. Consider for instance the term $(\lambda y.(\lambda z.z)y)x$ of example Example 1.1.8, it has the garbage-free reduction graph depicted in Figure 1.3 (compare to Figure 1.1). Note that the only place in this graph where a single garbage-free step consists of more than one raw step is where $((\lambda z.z)y)\langle y := x\rangle \xrightarrow[xap]{} ((\lambda z.z)\langle y := x\rangle)(y\langle y := x\rangle) \xrightarrow[gc]{} (\lambda z.z)(y\langle y := x\rangle)$.

1.2.6 Theorem (PSN for $\lambda x{\downarrow}gc$). *Pure terms that are $\xrightarrow[\beta]{}$-strongly normalising are also strongly normalising for $\xrightarrow[bx{\downarrow}gc]{}$.*

Proof. We will show that each $\xrightarrow[bx{\downarrow}gc]{}\!\!\!\!\rightarrow$-reduction corresponds to a $\xrightarrow[\beta]{}\!\!\!\!\rightarrow$-reduction of comparable length, hence there are no infinite $\xrightarrow[bx{\downarrow}gc]{}\!\!\!\!\rightarrow$-reductions for terms that are strongly normalising for $\xrightarrow[\beta]{}$; the proof is similar to that of Theorem 1.1.17. Assume M is pure and strongly normalising for $\xrightarrow[\beta]{}$. Since M is pure it has no $\xrightarrow[xgc]{}$-redexes and every $\xrightarrow[bx{\downarrow}gc]{}\!\!\!\!\rightarrow$-reduction starting with M, whether finite or not, is of the form $M \equiv M_0 \xrightarrow[b]{} M_1 \xrightarrow[xgc]{}\!\!\!\!\rightarrow M_2 \xrightarrow[b]{} M_3 \xrightarrow[xgc]{}\!\!\!\!\rightarrow \cdots$ where the "$\xrightarrow[xgc]{}\!\!\!\!\rightarrow$" segments are really $\underset{gc}{\twoheadrightarrow} \cdot (\xrightarrow[x]{} \cdot \underset{gc}{\twoheadrightarrow}) \cdots (\xrightarrow[x]{} \cdot \underset{gc}{\twoheadrightarrow})$ sequences. From such a sequence we can construct a $\xrightarrow[\beta]{}$-sequence as follows, from left to right:

$$\begin{tikzcd}
M \arrow[r, equal] & M_0 \arrow[r, "b"'] \arrow[d, equal] \arrow[dr, phantom, "(*)"] & M_1 \arrow[r, two heads, "xgc"'] \arrow[d, dotted, two heads, "x"'] \arrow[dr, phantom, "(**)"] & M_2 \arrow[r, "b"'] \arrow[d, dotted, two heads, "x"'] \arrow[dr, phantom, "(*)"] & M_3 \arrow[r, two heads, "xgc"'] \arrow[d, dotted, two heads, "x"'] \arrow[dr, phantom, "(**)"] & \cdots \\
& \downarrow_x(M_0) \arrow[r, dotted, "+", "\beta"'] & \downarrow_x(M_1) \arrow[r, dotted, equal] & \downarrow_x(M_2) \arrow[r, dotted, "+", "\beta"'] & \downarrow_x(M_3) \arrow[r, dotted, equal] & \cdots
\end{tikzcd}$$

where $(*)$ is Lemma 1.1.16.2 and $(**)$ is Proposition 1.1.13.1. Since the lower reduction is finite and $\xrightarrow[xgc]{}$SN, the upper one must also be finite. $\square$

Now we are ready to make garbage collection explicit again. In order to distinguish between the degrees of 'garbage reductions' such that we can identify the garbage collection overhead, we define two mutually disjoint classes of reductions: garbage-reductions and reductions outside garbage.

1.2.7 Definitions. We subdivide the reduction relation $\xrightarrow[bxgc]{}$ into two mutually disjoint parts.

1. *Garbage-reduction* is the contextual closure of the reduction generated by:

   - If $N \xrightarrow[bxgc]{} N'$ and $x \notin \mathrm{fv}(\downarrow_{gc}(M))$ then $M\langle x := N\rangle \xrightarrow[bxgc]{} M\langle x := N'\rangle$ is garbage-reduction.
   - if $x \notin \mathrm{fv}(\downarrow_{gc}(MN))$ then $(MN)\langle x := P\rangle \xrightarrow[bxgc]{} (M\langle x := P\rangle)(N\langle x := P\rangle)$ is garbage-reduction,
   - if $x \notin \mathrm{fv}(\downarrow_{gc}(\lambda y.M))$ then $(\lambda y.M)\langle x := N\rangle \xrightarrow[bxgc]{} \lambda y.M\langle x := N\rangle$ is garbage-reduction,
   - if $x \notin \mathrm{fv}(M)$ then $M\langle x := N\rangle \xrightarrow[bxgc]{} M$ is garbage-reduction,

   Note the use of $\mathrm{fv}(\downarrow_{gc}(\cdot))$ to ensure that for instance $(x\langle y := z\rangle x)\langle z := M\rangle \xrightarrow[x]{} (x\langle y := z\rangle\langle z := M\rangle)(x\langle z := M\rangle)$ is garbage-reduction.

2. Reduction *outside garbage* is any reduction that is not garbage-reduction; this is equivalent to saying that the contracted redex has no descendant in the $\xrightarrow[gc]{}$-normalform.

With this we can prove the following by induction on the structure of terms.

1.2.8 Propositions.

1. If $M \xrightarrow[\text{bxgc}]{} N$ is garbage-reduction then $\downarrow_{gc}(M) \equiv \downarrow_{gc}(N)$.

2. If $M \xrightarrow[\text{bxgc}]{} N$ is outside garbage then $\downarrow_{gc}(M) \xrightarrow[\text{bx}\downarrow\text{gc}]{} \downarrow_{gc}(N)$.

1.2.9 Definition. We say N *is body of a substitution in* M if for some P, x, $P\langle x := N\rangle$ is a subterm of M. The predicate $\mathrm{subSN}(M)$ should be read to be *all bodies of substitutions in* M *are strongly normalising for* $\xrightarrow[\text{bxgc}]{}$*-reduction.*

The following lemma expresses our intuition about garbage-reduction.

1.2.10 Lemmas.

1. If $\mathrm{subSN}(M)$ and $M \xrightarrow[\text{bxgc}]{} N$ is garbage-reduction, then $\mathrm{subSN}(N)$.

2. If $\mathrm{subSN}(M)$ then M is strongly normalising for garbage-reduction.

Proof. 1 Induction on the structure of M. We treat the case $M\langle x := N\rangle \xrightarrow[\text{bxgc}]{} M\langle x := N'\rangle$ where $x \notin \mathrm{fv}(\downarrow_{gc}(M))$. Then bodies of substitutions in N' are strongly normalising for $\xrightarrow[\text{bxgc}]{}$ since N' is a reduct of N which is strongly normalising for $\xrightarrow[\text{bxgc}]{}$ by $\mathrm{subSN}(M\langle x := N\rangle)$.

2 Define two interpretations for subSN-terms M. For terms M such that $\mathrm{subSN}(M)$, let $h_1(M)$ be the maximum length of $\xrightarrow[\text{bxgc}]{}$ inside garbage reduction paths of M. Since there are only finitely many substitutions in M and all bodies of these substitutions are strongly normalising by $\mathrm{subSN}(M)$, $h_1(M)$ exists. Define h_2 by the following:

$$\begin{aligned}
h_2(x) &= 1 \\
h_2(MN) &= h_2(M) + h_2(N) + 1 \\
h_2(\lambda x.M) &= h_2(M) + 1 \\
h_2(M\langle x := N\rangle) &= h_2(M) \times 2 \qquad \text{if } x \notin \mathrm{fv}(\downarrow_{gc}(M)) \\
h_2(M\langle x := N\rangle) &= h_2(M) \times (h_2(N) + 1) \qquad \text{if } x \in \mathrm{fv}(\downarrow_{gc}(M))
\end{aligned}$$

By straightforward induction on the structure of terms we can show that if $M \xrightarrow[\text{bxgc}]{} N$ is garbage reduction then $h_1(M) > h_1(N)$ and $h_2(M) = h_2(N)$. Hence garbage reduction is strongly normalising for subSN-terms. □

1.2.11 Definition. Define for all terms M, $\#\mathrm{gf}(M)$ to be the maximum length of garbage-free reduction paths starting in $\downarrow_{\mathrm{gc}}(M)$. Note that $\#\mathrm{gf}(M)$ can be infinite as it is for $(\lambda x.xx)(\lambda x.xx)$.

1.2.12 Theorem. *If* $\#\mathrm{gf}(M) < \infty$ *and* $\mathrm{subSN}(M)$ *then* M *is strongly normalising for* $\xrightarrow[\mathrm{bxgc}]{}$*-reduction.*

Proof. We use induction on $\#\mathrm{gf}(M)$.

Base case $\#\mathrm{gf}(M) = 0$. Then by Proposition 1.2.8.2 reduction paths of M cannot contain reductions outside garbage and hence by Lemma 1.2.10.2 they are finite.

Induction hypothesis Suppose $\#\mathrm{gf}(M) > 0$ and we already know that if $\#\mathrm{gf}(M') < \#\mathrm{gf}(M)$ and $\mathrm{subSN}(M')$ then M' is strongly normalising for $\xrightarrow[\mathrm{bxgc}]{}$ (IH1). Suppose there exists an infinite reduction path

$$M \equiv M_0 \xrightarrow[\mathrm{bxgc}]{} M_1 \xrightarrow[\mathrm{bxgc}]{} M_2 \xrightarrow[\mathrm{bxgc}]{} M_3 \xrightarrow[\mathrm{bxgc}]{} \cdots$$

By Lemma 1.2.10.2 there is m such that $M \xrightarrow[\mathrm{bxgc}]{}\!\!\!\!\!\!\rightarrow M_m$ is garbage-reduction and $M_m \xrightarrow[\mathrm{bxgc}]{} M_{m+1}$ is reduction outside garbage. Then by Proposition 1.2.8, $\#\mathrm{gf}(M_{m+1}) < \#\mathrm{gf}(M_m)$ and by Lemma 1.2.10.1, $\mathrm{subSN}(M_m)$. Now we prove by induction on the structure of M_m that also $\mathrm{subSN}(M_{m+1})$; then by IH1 we are done. Call the new induction hypothesis IH2. We treat some typical cases:

Case $M_m \equiv (N_1N_2)\langle x := P\rangle \xrightarrow[\mathrm{bxgc}]{} (N_1\langle x := P\rangle)(N_2\langle x := P\rangle) \equiv M_{m+1}$, where $x \in \mathrm{fv}(\downarrow_{\mathrm{gc}}(N_1N_2))$. Now bodies of substitutions in $(N_1\langle x := P\rangle)(N_2\langle x := P\rangle)$ are strongly normalising since they are also bodies of substitutions in $(N_1N_2)\langle x := P\rangle$ and $\mathrm{subSN}((N_1N_2)\langle x := P\rangle)$.

Case $M_m \equiv (\lambda x.N)P \xrightarrow[\mathrm{bxgc}]{} N\langle x := P\rangle \equiv M_{m+1}$. Bodies of substitutions in N and P are strongly normalising since they are also bodies of substitutions in $(\lambda x.N)P$. Furthermore $\#\mathrm{gf}(P) < \#\mathrm{gf}((\lambda x.N)P)$, hence by IH1 P is strongly normalising, thus $\mathrm{subSN}(N\langle x := P\rangle)$.

Case $M_m \equiv N\langle x := P\rangle \xrightarrow[\mathrm{bxgc}]{} N\langle x := P'\rangle \equiv M_{m+1}$ where $P \xrightarrow[\mathrm{bxgc}]{} P'$. Then P is strongly normalising for $\xrightarrow[\mathrm{bxgc}]{}$ since $\mathrm{subSN}(M_m)$, hence also P' is strongly normalising.

Case $M_m \equiv NP \xrightarrow[\mathrm{bxgc}]{} N'P \equiv M_{m+1}$ where $N \xrightarrow[\mathrm{bxgc}]{} N'$. Then $\mathrm{subSN}(N')$ by IH2, hence also $\mathrm{subSN}(N'P)$. $\square$

1.2.13 Corollary (PSN for λxgc). $\xrightarrow[\text{bxgc}]{}$ *PSN of* $\xrightarrow[\beta]{}$.

Proof. **Case** $\Rightarrow$. If M is pure then $\mathrm{subSN}(M)$ and if M is strongly normalising for $\xrightarrow[\beta]{}$-reduction then by Theorem 1.2.6, $\#\mathrm{gf}(M) < \infty$. Now use Theorem 1.2.12.

Case $\Leftarrow$. By Proposition 1.1.15.2, infinite $\xrightarrow[\beta]{}$-reductions induce infinite $\xrightarrow[\text{bxgc}]{}$-reductions. □

1.2.14 Remark. Strictly speaking, the above proof is not direct, however, it is easily modified into a direct one since the contradiction is not used in an essential way. An example of the (more roundabout) direct formulation can be found in the proof of Corollary 2.2.5.

The following corollary characterises which arbitrary terms of Λx are SN for $\xrightarrow[\text{bxgc}]{}$-reduction.

1.2.15 Corollary. *A* λxgc-*term* M *is SN for* $\xrightarrow[\text{bxgc}]{}$*-reduction if and only if for all subterms* N *of* M*:* $\downarrow_x(N)$ *is SN for* $\xrightarrow[\beta]{}$.

Proof. The only if part is easy. We prove the if part by induction on the maximal depth of nestings of substitutions in M. If the maximal depth of nestings is 0 then use Corollary 1.2.13.

Suppose that the maximal depth of nestings of substitutions is $n+1$. By the induction hypothesis, for all bodies N of substitutions in M: N is SN for $\xrightarrow[\text{bxgc}]{}$-reduction; therefore, $\mathrm{subSN}(M)$. After one garbage-free reduction step $M \xrightarrow[\text{bx}\downarrow\text{gc}]{} M'$, every garbage-free reduction path of M' will correspond to a $\xrightarrow[\beta]{}$-reduction path of $\downarrow_x(M)$ similar to the proof of Theorem 1.2.6. Therefore, $\#\mathrm{gf}(M) < \infty$. Now by Theorem 1.2.12, M is SN for $\xrightarrow[\text{bxgc}]{}$-reduction. □

1.2.16 Remark. Finally we mention a positive result: adding the rewrite rule

$$M\langle x := N\rangle\langle y := P\rangle \rightsquigarrow''' M\langle x := N\langle y := P\rangle\rangle \quad \textit{if } x \in \mathrm{fv}(\downarrow_{\mathrm{gc}}(M)) \textit{ and } y \notin \mathrm{fv}(M)$$

does not break PSN; see Bloo and Geuvers (1996) for details. Intuitively the reason for this is that if $x \in \mathrm{fv}(\downarrow_{\mathrm{gc}}(M))$ then in some $\xrightarrow[\text{bxgc}]{}$-reduct of $M\langle x := N\rangle\langle y := P\rangle$, indeed a subterm $N\langle y := P\rangle$ will occur, and accelerating the occurrence of this subterm can do no harm. In the rewrite rules discussed in Remark 1.2.1 substitutions are being introduced that cannot occur in a $\xrightarrow[\text{bxgc}]{}$ reduction path of $M\langle x := N\rangle\langle y := P\rangle$ and this is what breaks PSN, since then an infinite loop can be generated by composing substitutions with a redex inside and distributing over that redex.

1.3 Discussion

Several other techniques exist for proving confluence and PSN. The first is normally not difficult (we return to this in later chapters) but PSN is often hard. We have briefly mentioned the 'minimal derivation' method which was first used by Benaissa, Briaud, Lescanne and Rouyer-Degli (1995), however, a much more general method of Bloo and Geuvers (1996) uses the powerful *recursive path ordering* technique (invented by the name of "simplification orderings" by Dershowitz 1979) which seems to have great promise once it can be systematised and explained.

Chapter 2

Explicit Binding

In the last five years (since $\lambda\sigma$ of Abadi, Cardelli, Curien and Lévy 1991) many calculi of explicit substitution have been proposed in the literature. It is disappointing that the actual rules vary greatly, giving the calculi apparently different operational properties. The most notable difference is between the techniques used to represent *binding* of variables.

In this chapter we will argue for a large collection of these calculi that the differences are superficial in that reductions are essentially preserved, whence we can deduce that not only confluence but also normalisation is preserved for any 'reasonable' (in a precise sense) representation of binding.

We commence in Section 2.1 with a summary of de Bruijn's (1972) "namefree" calculus where natural numbers, called "indices," provide a syntactic representation of binding; we also show how to translate between namefree and ordinary λ-terms. (This makes heavy use of the "sequence" notation summarised in Appendix A.2.) We proceed in Section 2.2 with a survey of published substitution calculi based on de Bruijn indices and show how the translations can be shown to essentially preserve reduction length and term size between most of them (namely λx and each of $\lambda\upsilon$, λs, λt, and $\lambda\chi$), making all the published PSN results equivalent. Finally, we discuss in Section 2.1 the similarities and variations among these calculi and summarise the results.

The chapter covers the material of Rose (1996a) as well as summarises the syntactic properties of many published calculi (Abadi et al. 1991, Benaissa, Briaud, Lescanne and Rouyer-Degli 1995, de Bruijn 1972, Kamareddine and Ríos 1996, Lescanne 1994).

2.1 Namefree λ-Calculus

This section summarises the λβ-version of de Bruijn's (1972) *namefree* notation for λ-calculus. This is useful for observing things such as the number of variables in use, and many implementations use concepts from namefree λ-calculus to avoid the complications of name conflicts.

The namefree calculus was invented by de Bruijn as a variation of λ-calculus which was "efficient for automatic formula manipulation" in the system AUTOMATH (Nederpelt, Geuvers and de Vrijer 1994).

We exactly reproduce the 'pure' subset of de Bruijn's calculus, *i.e.*, without constants and δ-rules. However, we have used a notation which is consistent with the rest of this tutorial.

2.1.1 Definition (λNF). The λNF-terms are defined inductively by

$$a \ ::= \ n \mid \lambda a \mid ab$$

where the letters abc range over λNF-terms and n over natural numbers. β_{NF}-*reduction*, $\xrightarrow[\beta_{\mathrm{NF}}]{}$ is defined by the relation:

$$(\lambda a)b \to a[b \cdot \iota] \qquad (\beta_{\mathrm{NF}})$$

where the substitution derivor $\_[\_]$ is defined by

$$\begin{aligned} n[\phi] &= \phi(n) \\ (\lambda a)[\phi] &= \lambda \ (a[1 \cdot ((\iota/1) \circ \phi)]) \\ (ab)[\phi] &= (a[\phi]) \ (b[\phi]) \end{aligned}$$

using the sequence notations of Notation A.2.1 to define substitution; ϕ is always an infinite sequence.

The λNF-calculus is a very elegant way to get rid of the troublesome variable clash and capture problems. Its only weakness is the treatment of free variables: any index that is larger than the number of λs outside it is 'free', however, its identity is still determined by the 'ghost λ' to which it belongs. This is reflected by the definition of free variables which is $\mathrm{fv}(n) = \{n\}$, $\mathrm{fv}(\lambda a) = \{\, n \mid n+1 \in \mathrm{fv}(a) \,\}$, and $\mathrm{fv}(ab) = \mathrm{fv}(a) \cup \mathrm{fv}(b)$.

2.1.2 Example. The I, K, and S combinators are written $\lambda 1$, $\lambda\lambda 2$, and $\lambda\lambda\lambda 31(21)$ in λNF. The reduction of Example A.3.7 becomes $(\lambda(\lambda 1)1)1 \overset{\beta_{\mathrm{NF}}}{\underset{\beta_{\mathrm{NF}}}{\rightrightarrows}} (\lambda 1)1 \xrightarrow[\beta_{\mathrm{NF}}]{} 1$. Notice how the outermost 1 is free.

2.1.3 History. De Bruijn uses $S(\phi, a)$ to denote the substitution metanotion (our $a[\phi]$ with reversed ϕ), and τ_h for ι/h.

It is possible to relate Λ and ΛNF, of course: the following inference system does this, assuming the list of free variables is ρ (corresponding to the sequence of 'ghost λs'):

2.1.4 Definition (translation, λNF/λ). The closed λNF-term a projects onto the λ-term M, notation $a \rhd\!\!- M$, if we can prove $\epsilon \vdash a \rhd\!\!- M$ in the following system:

$$\rho \vdash n \rhd\!\!- x \qquad \rho(n) = x$$

$$\frac{x \cdot \rho \vdash a \rhd\!\!- M}{\rho \vdash \lambda a \rhd\!\!- \lambda x.M} \quad x \notin \rho$$

$$\frac{\rho \vdash a \rhd\!\!- M \qquad \rho \vdash b \rhd\!\!- N}{\rho \vdash ab \rhd\!\!- MN}$$

This generalises to open terms by quantifying over the list of free variable names.

2.2 Explicit Binding Variations for Explicit Substitution

In this section we give translations between λx(gc) and the other explicit substitution calculi and show that the translations are *strict conservative extensions* from which preservation of reductions follows (and hence PSN since λx(gc) has PSN).

We start by doing it for $\lambda\upsilon$ of Benaissa et al. (1995). This suggests a definition of the notion of *strict conservative extension*. We then use this idea to relate the other calculi.

2.2.1 Definition ($\lambda\upsilon$). See Figure 2.1 (or Benaissa et al. 1995).

An example reduction is shown in Figure 2.2 for $\lambda\upsilon$ and λx, where we have indicated the translation $\rhd\!\!\underset{\upsilon}{-}$. Since $\lambda\upsilon$ uses de Bruijn indices, the translation contains $\rhd\!\!-$ of Definition 2.1.4. The needed extension is quite mechanically derived from the need for the translation to be correct: reasoning in the sequence algebra gives that $\Uparrow$ with a shift substitution removes a variable from the scope wheras similarly $\Uparrow$ with a real substitution inserts one.

Terms. The $\lambda\upsilon$-*terms* are defined inductively by

$$
\begin{aligned}
a, b, c \in \mathit{Terms} &::= \underline{n} \mid \lambda a \mid ab \mid a[s] \\
s, t \in \mathit{Subst} &::= a/ \mid \Uparrow(s) \mid \uparrow
\end{aligned}
$$

where nm range over naturals.

Reduction.

$$
\begin{aligned}
(\lambda a)b &\to a[b/] && \text{(B)} \\
(\lambda a)[s] &\to \lambda(a[\Uparrow(s)]) && \text{(Lambda)} \\
(ab)[s] &\to a[s]\, b[s] && \text{(App)} \\
\underline{1}[a/] &\to a && \text{(FVar)} \\
\underline{n+1}[a/] &\to \underline{n} && \text{(RVar)} \\
\underline{1}[\Uparrow(s)] &\to \underline{1} && \text{(FVarLift)} \\
\underline{n+1}[\Uparrow(s)] &\to \underline{n}[s][\uparrow] && \text{(RVarLift)} \\
\underline{n}[\uparrow] &\to \underline{n+1} && \text{(VarShift)}
\end{aligned}
$$

The substitution reduction without (B) is denoted $\xrightarrow[\upsilon]{}$; the union of it and (B) is $\xrightarrow[\lambda\upsilon]{}$.

Figure 2.1: $\lambda\upsilon$.

2.2.2 Definition (translation, $\lambda\upsilon/\lambda x$). For a $\lambda\upsilon$-term a and a λx-term M we write $a \mathrel{\triangleright\!\!\!-_{\upsilon}} M$ iff $\epsilon \vdash a \mathrel{\triangleright\!\!\!-_{\upsilon}} M$, where $\triangleright\!\!\!-_{\upsilon}$ is obtained by extending $\triangleright\!\!\!-$ with the two rules

$$
\frac{\rho\backslash n \cdot x \cdot \rho/n \vdash a \mathrel{\triangleright\!\!\!-_{\upsilon}} M \quad \rho/n \vdash b \mathrel{\triangleright\!\!\!-_{\upsilon}} N}{\rho \vdash a[\Uparrow^n(b/)] \mathrel{\triangleright\!\!\!-_{\upsilon}} M\langle x := N\rangle} \qquad x \notin \rho,\ n \geq 0
$$

$$
\frac{\rho\backslash n \cdot \rho/(n+1) \vdash a \mathrel{\triangleright\!\!\!-_{\upsilon}} M}{\rho \vdash a[\Uparrow^n(\uparrow)] \mathrel{\triangleright\!\!\!-_{\upsilon}} M} \qquad n \geq 0
$$

That this corresponds closely to λx is seen as follows:

2.2.3 Propositions. 1. $\triangleright\!\!\!-_{\upsilon}$ projects any $\xrightarrow[\lambda\upsilon]{}\!\!\!\!\rightarrow$-reduction into one with $\xrightarrow[\text{bxgc}]{}\!\!\!\!\rightarrow$.

2. $\triangleright\!\!\!-_{\upsilon}$ projects any $\xrightarrow[\text{B}]{}$-reduction into one with $\xrightarrow[\text{b}]{}$.

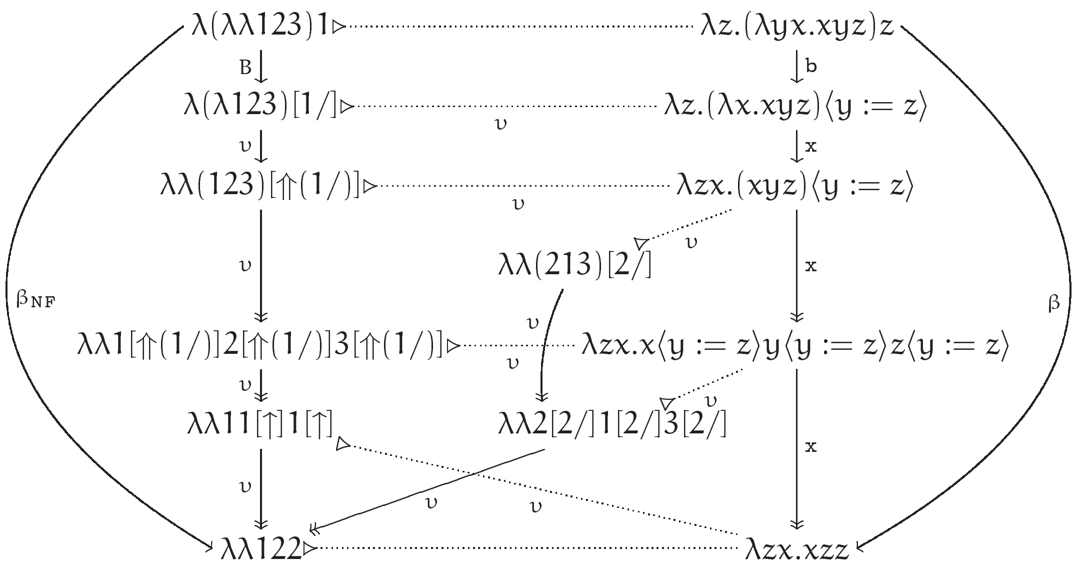


Figure 2.2: A reduction in $\lambda\upsilon$ (left) and λx (right).

3. $\triangleright_{\overline{\upsilon}}$ projects any $\twoheadrightarrow_{\upsilon}$-reduction into one with $\twoheadrightarrow_{x}$.

Proof. First of all, it is easy to see from Definition 2.2.2 that $\triangleright_{\overline{\upsilon}}$ is a function. For this reason we will in this proof use the notation $\overline{a\rho}$ for the unique λx-term M satisfying $\rho \vdash a \triangleright_{\overline{\upsilon}} M$.

1 is clearly a direct consequence of 2 and 3 which we will prove by induction over the structure of $\lambda\upsilon$-terms, using the definition to investigate each $\lambda\upsilon$-rewrite rule which must appear at a subterm with some free variable assignments. Essentially we have to prove that for any (non-repeating) variable list ρ that $\overline{a\rho} \to \overline{b\rho}$. 2 has the base case

$$\overline{(\lambda a)b\ \rho} \equiv (\lambda x.\overline{a((x)\cdot\rho)})\ \overline{b\rho} \xrightarrow[\text{bxgc}]{} \overline{a((x)\cdot\rho)}\ \langle x := \overline{b\rho}\rangle \equiv \overline{a[b/]\ \rho}$$

since $x \notin \rho$.

For 3 we include just the two cases for (Lambda) (the full proof is in Rose 1996b, Prop.3.3.7).

Case (Lambda) with real substitution.

$$\overline{(\lambda a)[\Uparrow^n(b/)]\ \rho}$$

$$
\begin{aligned}
&\equiv \overline{(\lambda a)\ ((\rho\backslash n)\cdot(x)\cdot(\rho/n))}\langle x := \overline{b\ (\rho/n)}\rangle\\
&\equiv \big(\lambda y.\overline{a\ ((y)\cdot(\rho\backslash n)\cdot(x)\cdot(\rho/n))}\big)\langle x := \overline{b(\rho/n)}\rangle\\
&\xrightarrow[x]{} \lambda y.\overline{a\ ((y)\cdot(\rho\backslash n)\cdot(x)\cdot(\rho/n))}\langle x := \overline{b(\rho/n)}\rangle)\\
&\equiv \lambda y.\overline{a\ ((y)\cdot(\rho\backslash n)\cdot(x))\cdot(((y)\cdot\rho)/n+1)}\langle x := \overline{b((y)\cdot\rho)/n+1)}\rangle\\
&\equiv \lambda y.\overline{(a[\Uparrow^{n+1}(b/)])\ ((y)\cdot\rho)}\\
&\equiv \overline{\lambda(a[\Uparrow^{n+1}(b/)])\ \rho}
\end{aligned}
$$

Case (Lambda) with shift-substitution.

$$
\begin{aligned}
\overline{(\lambda a)[\Uparrow^{n}(\uparrow)]\ \rho} &\equiv \overline{(\lambda a)\ ((\rho/n)\cdot(\rho/n+1))}\\
&\equiv \lambda x.\overline{a\ ((x)\cdot(\rho/n)\cdot(\rho/n+1))}\\
&\equiv \lambda x.\overline{a\ (((x)\cdot(\rho/n))\cdot((x)\cdot\rho)/n+2)}\\
&\equiv \lambda x.\overline{a[\Uparrow^{n+1}(\uparrow)]\ ((x)\cdot\rho)}\\
&\equiv \overline{\lambda(a[\Uparrow^{n+1}(\uparrow)])\ \rho}
\end{aligned}
$$

It is interesting to observe that the $\lambda\upsilon$-reduction steps that correspond to no λx-steps after translations are (Lambda) and (App) with $s = \uparrow$, (FVarLift), (RVarLift), and (VarShift). ☐

The above suffices to show the desired properties of $\lambda\upsilon$, namely that it is a conservative extension of the $\lambda\beta$-calculus that preserves strong normalisation:

2.2.4 Theorem. *Given $\lambda\upsilon$-terms a, b and pure λ-terms M, N such that $a \triangleright_{\upsilon} \cdot \twoheadrightarrow_{x} M$ and $b \triangleright_{\upsilon} \cdot \twoheadrightarrow_{x} N$. Then $a \xrightarrow[\lambda\upsilon]{+} b$ if and only if $M \xrightarrow[\beta]{+} N$.*

Proof. Consider any $\lambda\upsilon$-reduction sequence. It must correspond to a λx-sequence as follows by Proposition 2.2.3.2 and Proposition 2.2.3.3:

$$
\begin{tikzcd}
\blacksquare \arrow[r, two heads, "\upsilon"'] \arrow[d, "\upsilon"'] & \blacksquare \arrow[r, "B"'] \arrow[d, "\upsilon"'] & \blacksquare \arrow[r, two heads, "\upsilon"'] \arrow[d, "\upsilon"'] & \cdots \arrow[r, two heads, "\upsilon"'] & \blacksquare \arrow[r, "B"'] \arrow[d, "\upsilon"'] & \blacksquare \arrow[r, two heads, "\upsilon"'] \arrow[d, "\upsilon"'] & \blacksquare \arrow[d, "\upsilon"'] \\
\blacksquare \arrow[r, two heads, "x"'] & \blacksquare \arrow[r, "b"'] & \blacksquare \arrow[r, two heads, "x"'] & \cdots \arrow[r, two heads, "x"'] & \blacksquare \arrow[r, "b"'] & \blacksquare \arrow[r, two heads, "x"'] & \blacksquare
\end{tikzcd}
$$

This is thus finite exactly when the original sequence was finite because $\xrightarrow[\upsilon]{}$SN. Since λx has PSN by Corollary 1.2.13 we are done. (Original indirect proof by Benaissa et al. 1995, section 5). ☐

2.2.5 Corollary (PSN, $\lambda\upsilon$). *$\lambda\upsilon$ PSN of $\lambda\beta$*

Proof. Direct consequence of Proposition 2.2.3.1 and Corollary 1.2.13. □

2.2.6 Comparison (on minimal derivations). Comparing this proof with the original proof of Benaissa et al. is interesting. The translation to λx has eliminated the need for considering 'minimal derivations' and positions that are 'internal/external' to substitutions. In the PSN proof for λx (Bloo and Rose 1995) this is replaced by the notion of 'inside garbage' which is clearly a stronger requirement in that some 'internal' positions will not be inside garbage.

Notice that except for the translation itself no particular concrete properties of $\lambda\upsilon$ were used in the above proof, only abstract reduction properties. This suggests that we can generalise the proof to any explicit substitution calculus with a 'well-behaved' notion of 'explicit binding.' In the remainder of this section we first define such a notion in general, then refine it to include the constraints above in abstract form, and finally give proofs of PSN for the other calculi mentioned above.

2.2.7 Definition (explicit binding). An *explicit binding principle* is a projection from a TRS into a reduction on a λ-calculus with names.

So Definition 2.1.4 and Definition 2.2.2 are binding principles. The crucial property for PSN is that there is a distinguished 'β-like' subreduction that is repeated as frequently as β-reduction. This we use to define a 'β-strict conservative extension of λx' which is what we need to obtain the desired PSN results.

2.2.8 Definition (β-strict conservative extension). The TRS $\rightsquigarrow$ is a *strict explicit binding extension of* λx if $\rightsquigarrow = \underset{\mathrm{B}}{\rightsquigarrow} \cup \underset{\mathrm{x}}{\rightsquigarrow}$ such thatthe following conditions are satisfied:

1. $\rhd\!\!-$ projects from $\rightsquigarrow\!\!\!\!\twoheadrightarrow$ to $\underset{\mathrm{bxgc}}{\twoheadrightarrow}$.

2. $\underset{\mathrm{x}}{\rightsquigarrow}$SN and $\rhd\!\!-$ projects from $\underset{\mathrm{x}}{\rightsquigarrow\!\!\!\!\twoheadrightarrow}$ to $\underset{\mathrm{x}}{\twoheadrightarrow}$.

3. $\rhd\!\!-$ projects from $\overset{+}{\underset{\mathrm{B}}{\rightsquigarrow}}$ to $\overset{+}{\underset{\mathrm{B}}{\rightarrow}}$.

2.2.9 Theorem (PSN for strict conservative extension). *If $\rightsquigarrow$ is a β-strict conservative extension of* λx*, then* $\rightsquigarrow$PSN.

Now we apply this theorem to some other calculi with translations.

Terms. The λs-*terms* are defined inductively by

$$a, b \in \mathit{Terms} ::= n \mid ab \mid \lambda a \mid a\,\sigma^i\, b \mid \varphi^i_k\, a$$

Reduction.

$$
\begin{aligned}
(\lambda a)b &\to a\,\sigma^1\, b && (\sigma\text{-gen})\\
(\lambda a)\sigma^i\, b &\to \lambda(a\,\sigma^{i+1}\, b) && (\sigma\text{-}\lambda\text{-trans})\\
(a_1 a_2)\sigma^i\, b &\to (a_1\,\sigma^i\, b)(a_2\,\sigma^i\, b) && (\sigma\text{-app-trans})\\
n\,\sigma^i\, b &\to \begin{cases} n-1 & \text{if } n > i\\ \varphi^i_0\, b & \text{if } n = i\\ n & \text{if } n < i\end{cases} && (\sigma\text{-destr})\\
\varphi^i_k(\lambda a) &\to \lambda(\varphi^i_{k+1}\, a) && (\varphi\text{-}\lambda\text{-trans})\\
\varphi^i_k(a_1 a_2) &\to (\varphi^i_k\, a_1)(\varphi^i_k\, a_2) && (\varphi\text{-app-trans})\\
\varphi^i_k\, n &\to \begin{cases} n+i-1 & \text{if } n > k\\ n & \text{if } n \le k\end{cases} && (\varphi\text{-destr})
\end{aligned}
$$

The substitution reduction without (σ-gen) is denoted $\underset{s}{\to}$; the union of it and (σ-gen) is $\underset{\lambda s}{\to}$.

Figure 2.3: λs.

2.2.10 Definition (λs). See Figure 2.3 (or Kamareddine and Ríos 1995).

2.2.11 Definition (translation, $\lambda s/\lambda x$). For an λs-term a and a λx-term M we write $a \mathrel{\underline{\triangleright}_s} M$ iff $\epsilon \vdash a \mathrel{\underline{\triangleright}_s} M$, where

$$\frac{\rho\backslash(i-1)\cdot x\cdot\rho/(i-1) \vdash a \mathrel{\underline{\triangleright}_s} M \quad \rho/i \vdash b \mathrel{\underline{\triangleright}_s} N}{\rho \vdash a\,\sigma^i\, b \mathrel{\underline{\triangleright}_s} M\langle x := N\rangle} \qquad x \notin \rho$$

$$\frac{\rho\backslash k\cdot\rho/(k+i-1) \vdash a \mathrel{\underline{\triangleright}_s} M}{\rho \vdash \varphi^i_k a \mathrel{\underline{\triangleright}_s} M}$$

2.2.12 Theorem. λs *is a* β*-strict conservative extension of* λx.

Proof. $\underset{\lambda s}{\to}$SN proved by Kamareddine and Ríos (1995); the translation conditions proved analogously to for $\lambda\upsilon$. ☐

2.2.13 Corollary (PSN λs). λs *PSN of* $\lambda\beta$.

Proof. Immediate from being a β-strict conservative extension of λx (original indirect proof by Kamareddine and Ríos 1996). □

The next calculus uses *de Bruijn's levels*: fixed variable names but with explicit renaming (α-conversion).

2.2.14 Definition ($\lambda\chi$). See Figure 2.4 (or Lescanne and Rouyer-Degli 1995).

2.2.15 Theorem. $\lambda\chi$ *is a* β*-strict conservative extension of* λx.

Proof. $\underset{\lambda\chi}{\rightarrow}$SN proved by Lescanne and Rouyer-Degli (1995). Use $\underset{\alpha}{\twoheadrightarrow}$ as projection since we can reuse the x_i as λx-variables. □

2.2.16 Corollary (PSN $\lambda\chi$). $\lambda\chi$ *PSN of* $\lambda\beta$.

Proof. Immediate from being a β-strict conservative extension of λx (first, indirect, proof by Lescanne and Rouyer-Degli 1995). □

To demonstrate the versatility of garbage collection we now invent a variant of $\lambda\upsilon$ which is just as efficient but exploits knowledge about the de Bruijn representation of binding to do 'free' garbage collection. The idea is simple: we identify $n+1$ and $n[\uparrow]$ by internalising (VarShift).

2.2.17 Definition ($\lambda\upsilon 1$). See Figure 2.5.

2.2.18 Theorem. $\lambda\upsilon 1$ *is a* β*-strict conservative extension of* λxgc.

Proof. Similar to the $\lambda\upsilon$ proof with the corresponding projection. The difference is that now we actually need λxgc because (RVar) does a limited form of garbage collection! The case looks as follows:

Case (RVar). $\overline{a[\uparrow][b/]\ \rho} \equiv \overline{a[\uparrow](x \cdot \rho)}\ \langle x := \overline{b\rho}\rangle \equiv \overline{a\rho}\ \langle x := \overline{b\rho}\rangle \underset{gc}{\rightarrow} \overline{a\rho}$ with $x \notin \rho$.

□

2.2.19 Corollary (PSN $\lambda\upsilon 1$). $\lambda\upsilon 1$ *PSN of* $\lambda\beta$.

Finally, the recent λt of Kamareddine and Ríos (1996).

Terms. The $\lambda\chi$-*terms* are defined inductively by

$$a, b, c \in \mathit{Terms} ::= x_i \mid \lambda x_i \cdot a \mid ab \mid a[b/x_i]_j \mid \alpha(a, i, j)$$

where the ijk range over natural numbers (note that 'x' is *syntax* in this calculus, the level number is the suffix). Only terms that satisfy $a : \mathsf{Term}_1$ given by the following system are allowed:

$$\frac{}{x_i : \mathsf{Term}_{i+j+1}} \qquad \frac{a : \mathsf{Term}_i \quad b : \mathsf{Term}_i}{ab : \mathsf{Term}_i}$$

$$\frac{a : \mathsf{Term}_{i+1}}{\lambda x_i \cdot a : \mathsf{Term}_i} \qquad \frac{x_i : \mathsf{Term}_{i+j+1} \quad b : \mathsf{Term}_i}{a[b/x_i]_j : \mathsf{Term}_{i+j}}$$

Reduction.

$$\begin{array}{rcll}
(\lambda x_i \cdot a)b & \to & a[b/x_i]_0 & \text{(B)} \\
(\lambda x_{i+j+1} \cdot a)[b/x_i]_j & \to & \lambda x_{i+j} \cdot (a[b/x_i]_{j+1}) & \text{(Lambda)} \\
(a\ b)[c/x_i]_j & \to & a[c/x_i]_j\ b[c/x_i]_j & \text{(App)} \\
x_{i+k+1}[a/x_i]_j & \to & x_{i+k} & (\text{Var}_{>}) \\
x_i[a/x_{i+k+1}]_j & \to & x_i & (\text{Var}_{<}) \\
x_i[a/x_i]_j & \to & \alpha(a, i, j) & (\text{Var}_{=}) \\
\alpha(ab, i, j) & \to & \alpha(a, i, j)\ \alpha(b, i, j) & (\alpha\text{-App}) \\
\alpha(\lambda x_{i+k} \cdot a, i, j) & \to & \lambda x_{i+k+j} \cdot \alpha(a, i, j) & (\alpha\text{-Lambda}) \\
\alpha(x_{i+k}, i, j) & \to & x_{i+j+k} & (\alpha\text{-Var}_{\geq}) \\
\alpha(x_i, i+k+1, j) & \to & x_i & (\alpha\text{-Var}_{<})
\end{array}$$

As usual we define $\twoheadrightarrow_{\lambda\chi}$ and $\rightarrow_{\chi}$; $\rightarrow_{\alpha}$ is the subrelation permitting only the four last rules.

Figure 2.4: $\lambda\chi$.

Terms. The $\lambda\upsilon 1$-*terms* are defined inductively by

$$a, b, c \in \mathit{Terms} ::= 1 \mid \lambda a \mid ab \mid a[s] \tag{2.1}$$
$$s, t \in \mathit{Subst} ::= a \mid \Uparrow(s) \mid \uparrow \tag{2.2}$$

Reduction.

$$(\lambda a)b \to a[b] \tag{B}$$
$$(\lambda a)[s] \to \lambda(a[\Uparrow(s)]) \tag{Lambda}$$
$$(ab)[s] \to a[s]\, b[s] \tag{App}$$
$$a[\uparrow][b] \to a \tag{Shift}$$
$$a[\uparrow][\Uparrow(s)] \to a[s][\uparrow] \tag{ShiftLift}$$
$$1[a] \to a \tag{Var}$$
$$1[\Uparrow(s)] \to 1 \tag{VarLift}$$

$\lambda\upsilon 1$-*reduction* includes all the rules (with a $\upsilon 1$-substitution calculus as usual).

Figure 2.5: $\lambda\upsilon 1$.

2.2.20 Definition (λt). See Figure 2.6 (or Kamareddine and Ríos 1996).

2.2.21 Definition (translation, λt/λx). For an λt-term a and a λx-term M we write $a \rhd_{\overline{t}} M$ iff $\epsilon \vdash a \rhd_{\overline{t}} M$, where

$$\frac{\rho\backslash(i-1)\cdot x \cdot \rho/(i-1) \vdash a \rhd_{\overline{t}} M \quad \rho \vdash b \rhd_{\overline{t}} N}{\rho \vdash a \,\varsigma^i\, b \rhd_{\overline{t}} M\langle x := N\rangle} \quad x \notin \rho$$

$$\frac{\rho\backslash k \cdot \rho/(k+1) \vdash a \rhd_{\overline{t}} M}{\rho \vdash \theta_k a \rhd_{\overline{t}} M}$$

2.2.22 Theorem. λt *is a* β*-strict conservative extension of* λx.

Proof. $\underset{\lambda t}{\to}$SN proved by Kamareddine and Ríos (1996); the rest completely analogously to the λs-proof. $\square$

Terms. λt-terms are defined inductively by

$$a, b, c \in \textit{Terms} ::= n \mid ab \mid \lambda a \mid a\,\varsigma^i\,b \mid \theta_k\,a$$

Reduction.

$$
\begin{aligned}
(\lambda a)b &\to a\,\varsigma^1\,b && (\varsigma\text{-gen})\\
(\lambda a)\varsigma^i\,b &\to \lambda(a\,\varsigma^{i+1}\,\theta_0(b)) && (\varsigma\text{-}\lambda\text{-trans})\\
(a_1 a_2)\varsigma^i\,b &\to (a_1\,\varsigma^i\,b)(a_2\,\varsigma^i\,b) && (\varsigma\text{-app-trans})\\
n\,\varsigma^i\,b &\to \begin{cases} n-1 & \text{if } n > i\\ b & \text{if } n = i\\ n & \text{if } n < i\end{cases} && (\varsigma\text{-destr})\\
\theta_k(\lambda a) &\to \lambda(\theta_{k+1}\,a) && (\theta\text{-}\lambda\text{-trans})\\
\theta_k(a_1 a_2) &\to (\theta_k\,a_1)(\theta_k\,a_2) && (\theta\text{-app-trans})\\
\theta_k\,n &\to \begin{cases} n+1 & \text{if } n > k\\ n & \text{if } n \le k\end{cases} && (\theta\text{-destr})
\end{aligned}
$$

The substitution reduction without (ς-gen) is denoted $\xrightarrow[t]{}$; the union of it and (ς-gen) is $\xrightarrow[\lambda t]{}$.

Figure 2.6: λt.

2.2.23 Corollary (PSN λt). *λt PSN of $\lambda\beta$.*

Proof. Immediate from being a β-strict conservative extension of λx (similar translation proof 'by hand' by Kamareddine and Ríos 1995). ☐

Finally we discuss why the above is not applicable to $\lambda\sigma$ which is fortunate since $\lambda\sigma$ *not* PSN of $\lambda\beta$.

2.2.24 Definition ($\lambda\sigma$). The $\lambda\sigma$-calculus is defined in Figure 2.7.

Comparing this to de Bruijn's Definition 2.1.1, it is easy to see it simply makes *everything* explicit: the syntactic $\_ \cdot \_$ and $\_ \circ \_$ realise the similarly named sequence operators, id is our ι, and $\uparrow$ is $\iota/1$. The reason $\lambda\sigma$ does not preserve PSN, observed by Melliès (1995), is simple *interference* between the two subrelations

Terms. The $\lambda\sigma$-terms $\Lambda\sigma$ are defined inductively by

$$
\begin{aligned}
a &::= 1 \mid \lambda a \mid ab \mid a[s] && \text{(terms)}\\
s &::= \mathsf{id} \mid \uparrow \mid a \cdot s \mid s \circ t && \text{(substitutions)}
\end{aligned}
$$

where the letters abc range over terms and st over substitutions (note: here the $\cdot$, $\circ$, and id are syntax).

Substitution generation. $\xrightarrow[\mathsf{Beta}]{}$ is the contextual closure of

$$(\lambda a)b \to a[b \cdot \mathsf{id}] \qquad \text{(Beta)}$$

Substitution elimination. $\xrightarrow[\sigma]{}$ is the contextual closure of the union of

$$
\begin{aligned}
1[\mathsf{id}] &\to 1 && \text{(VarId)}\\
1[a \cdot s] &\to a && \text{(VarCons)}\\
(ab)[s] &\to a[s]\ b[s] && \text{(App)}\\
(\lambda a)[s] &\to \lambda(a[1 \cdot (s \circ \uparrow)]) && \text{(Abs)}\\
(a[s])[t] &\to a[s \circ t] && \text{(Clos)}\\
\mathsf{id} \cdot s &\to s && \text{(IdL)}\\
\uparrow \circ\, \mathsf{id} &\to \uparrow && \text{(ShiftId)}\\
\uparrow \circ\, (a \cdot s) &\to s && \text{(ShiftCons)}\\
(a \cdot s) \circ t &\to a[t] \cdot (s \circ t) && \text{(Map)}\\
(s_1 \circ s_2) \circ s_3 &\to s_1 \circ (s_2 \circ s_3) && \text{(Assoc)}
\end{aligned}
$$

Figure 2.7: $\lambda\sigma$.

$\xrightarrow{\sigma}$ and $\xrightarrow{\mathsf{Beta}}$: the substitution subrelation can accidentally create a Beta-redex (*cf.* Figure 1.2).

2.2.25 Discussion. This section was mostly about how one can express existing de Bruijn calculi with explicit substitution by two components: λx and a renaming principle. However, why not start with de Bruijn's calculus as it is and make the substitution notion of Definition 2.1.1 explicit? In fact de Bruijn (1972, sec. 7) essentially suggests this in the observation that "In automatic formula manipulation it may be a good strategy to refrain from evaluating such $\tau_\ell(\langle\Sigma\rangle)$'s, but just store them as pairs $\ell, \langle\Sigma\rangle$, and go into (full or partial) evaluation only if necessary." (recall that $\tau_\ell(\langle\Sigma\rangle)$ is de Bruijn's notation for $(\iota/\ell) \circ \phi$).

The above calculi suggest why this is not so obvious: de Bruijn's Definition 2.1.1 of substitution involves (in our notation) *three* metanotations in the definition of $\_[\_]$:, namely, $\_ \cdot \_$, $\_/1$, and $\_ \circ \_$. The only one including them all is $\lambda\sigma$ where it gives problems because they do not constitute a complete system; the others restrict it in non-intuitive ways.

We believe our presentation has clarified this. Further work is needed along these lines, notably understanding the naming dimension sufficiently to parametrise different namings more clearly – possibly the work of Gordon (1993) is appropriate to this end.

2.3 Discussion

All the calculi we have studied here realise 'pure' explicit substitution, that is, *conservative extensions* of the λβ-calculus, consequently the discussed calculi were confluent and contextual (as λx). This constraint ensures that we can interpret the calculi into λβ proper and thus that they are *confluent* (on ground terms) as well as *weakly normalising.*

They still vary considerably, however. We conclude this section by a discussion of what seems to be the four principal distinctions:

Explicit binding? Does the calculus represent λ-binding implicitly or explicitly?

> Implicit binding is as in λx: reduction is defined modulo renaming of bound variables (usually called α-conversion) such that $\lambda x.x$ and $\lambda y.y$ are indistinguishable 'representatives' of the same term of the calculus. It also means that one must syntactically distinguish between free and bound variables of terms (free variables cannot be renamed without changing the meaning of terms).

With explicit binding, variables are encoded by terms of some term algebra. This has the advantage that the reduction may be defined without binding as a first order rewrite system. Obviously this means that free and bound variables are not syntactically distinguishable (since the coding must be known to distinguish).

This issue was the main focus of this chapter. An interesting consequence of our development is that the explicitness of binding does not change the operational behaviour in any essential way.

Preservation of β-strong normalisation (PSN)? To which extent is the calculus SN on terms that are β-SN? This issue has been the focus of much research since Melliès's (1995) counterexample to PSN for $\lambda\sigma$.

In this paper we show that translations between different explicit binding principles preserves PSN results, hence we prove several known PSN results equivalent.

Open terms? Does the system have a notion of open terms, *i.e.*, terms including metavariables denoting uninstantiated subterms? This is useful for reasoning about partly instantiated terms. Careful justification is needed in each case, however, as the notion of 'open term' depends on the *sorts* of the calculus. One such justification in calculi with a separate substitution sort is that it is useful to have confluence on terms with uninstantiated substitutions because this makes it possible to reason about reduction in arbitrary contexts of terms with free variables. The calculi with a notion of open terms are usually extended to obtain the property (for further details consult Curien, Hardin and Lévy 1996, Muñoz Hurtado 1996, Kamareddine and Ríos 1996).

Finite rule set? Is the system given as a finite set of proper rewrite rules or a rule schemas describing an infinite system? First-order finite rule systems can be implemented with a bound on the complexity of rewriting, for infinite systems this is not clear and thus infinite systems require careful justification that they are operationally useful. All the infinite systems considered here are infinite because they require matching of subpatterns such as '$\mathtt{i}+\mathtt{j}$' meaning that some number must be split in two nondeterministically requiring an infinity of (first order) rules for the possible splits.

All the above are as follows for the original explicit substitution calculus $\lambda\sigma$ and the calculi we have considered (✓ means yes, × no, and ? that the author has seen a claim but no proof):

Calculus	#rules	binding	PSN	open t.
$\lambda\sigma$	11	de Bruijn	×	($\lambda\sigma_{\Uparrow}$)
$\lambda\upsilon$	8	de Bruijn	✓	×
λs	∞	de Bruijn	✓	(λs_e?)
λt	∞	de Bruijn	✓	×
$\lambda\chi$	∞	direct	✓	×
λx	5	implicit	✓	×
$\lambda\upsilon 1$	7	de Bruijn	✓	×

The most interesting open questions in this area is whether a calculus of explicit substitution exists with PSN which is confluent for a meaningful notion of open terms and finite in addition to being a conservative extension of β-reduction.

We conjecture that the answer to this question is negative.

Chapter 3

Explicit Sharing

The only missing piece before we can claim that our substitution calculi are operationally faithful is that we have not argued that they have a notion of *space*.

In this chapter we do this: we study the *explicit sharing* dimension in section 3.1 by introducing sharing in general and for λ-calculus in particular, a major part of the latter is a comparison to Wadsworth's (1971) λ-graph reduction. In section 3.2 we synthesise explicit sharing and substitution into λxa with which we can model the *complexity of sharing* independently of the reduction strategy. This is the first system where the *reduction count is a realistic complexity measure*.

3.1 λ-graphs and Explicit Sharing

A central theme to this tutorial is the notion of *sharing*. We present λxa, a λ-calculus with explicit sharing, however, we attempt to express all properties in as general a form as possible.

Our *syntax* for sharing is based on *address labels*. The main idea behind this is that sharing is 'extra information' in the sense that two subterms are shared when they occupy the same memory; we indicate this by marking the two subterms, which should of course be identical, with the same address label. It is then realistic to imagine *simultaneous reduction* of all 'instances' of the shared subterm, and this is what we will do, thus this is a synthesis of the notion of *parallel reduction*[1] and that of *term graph rewriting* (TGR) of Barendregt,

[1] The notion of 'parallel reduction' has suffered the dubious fate of being regarded as 'computer science folklore', *i.e.*, its origin is unclear. We have it from Lévy (1978).

van Eekelen, Glauert, Kennaway, Plasmeijer and Sleep (1987). TGR was based on the λ-graph reduction of Wadsworth (1971), and indeed the λ-calculus we introduce is closely related to Wadsworth's reduction as we shall see. This section is restricted to 'acyclic' sharing as this is the only kind where a universally accepted formalism can be said to exist.

3.1.1 Definitions (addressed terms). Assume a set of terms $\mathbb{T}$ inductively defined by a single production $t ::= \cdots$.

1. The set of *adressed* $\mathbb{T}$*-terms* is defined inductively by

$$t \ ::= \ \cdots \ \mid \ t^{a}$$

   where abc range over an infinite set of 'addresses'.

2. The *addresses* occuring in an addressed term is written $\operatorname{addr}(t)$.

3. *Unraveling* is written $\triangle(t)$ and denotes the $\mathbb{T}$-term obtained by removing all the addresses in t.

These notions generalise in the obvious way to sets defined by several productions.

As the intention with addressed terms is that they should contain *sharing information* we need a way to express that the addresses of a term describe proper sharing, and a way to change an addressed term without destroying this.

3.1.2 Definitions (sharing terms). Assume a set of adressed $\mathbb{T}$-terms as above.

1. We write 't wfa' if t is an addressed term where all subterms with the same addresses are identical, *i.e.*,

$$\forall\, t_1^{a_1}, t_2^{a_2} \trianglerighteq t : a_1 = a_2 \Longrightarrow t_1 = t_2$$

   For t wfa we will write the *subterm of* t *at address* a as $t@a$ by convention $t@a \equiv \epsilon$ if $a \notin \operatorname{addr}(t)$, where ϵ is a new symbol.

2. The *updating* of t wfa with s wfa at address $a \notin \operatorname{addr}(s)$ is written $t[s^{a}]$ and is obtained by replacing all subterms $u^{a} \trianglerighteq t$ (if any) by s^{a}. By convention $t[\epsilon] \equiv t$ (to ensure $t[t@a] = t$ for all a).

3. The set of *sharing* $\mathbb{T}$*-terms*, notation $\mathbb{T}$ wfa, is the subset of the addressed $\mathbb{T}$-terms with wfa.

When a term is wfa we will say for each address in it that "the" subterm t^{a} "is" *shared* with address a.

3.1.3 Remark (graph reduction intuition). The wfa notion is intended to be intuitively similar to graph reduction, of course. Notice that a subterm can have many addresses, *e.g.*, $(t^a)^b$. Abstractly this just means that it is 'involved' in several sharing relationships. The restriction above means that these must be nested properly: the preterm $((t^a)^b)(t^b)$ is not wfa.

In fact, the address b in the term $((t^a)^b)$ corresponds very closely to an *indirection node* in a of graph reduction system: when an indirection node at address b points to the node in address a then all pointers to the indirection effectively point to the node at a but not vice versa.

Similarly, removal of an address corresponds to one of two things in the graph reduction terminology: Either *indirection elimination* when removing a in a subterm of an addressing $(t^a)^b$ 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 t^a before and t after). It corresponds to plain copying in all other contexts because the removed address then implicitly requires that an implicitly unique instance be used.

3.1.4 Proposition (sharing stable under updating). Given s, t wfa such that for all common addresses $a \in \mathrm{addr}(s) \cap \mathrm{addr}(t)$, $s@a = t@a$, and an address $c \notin \mathrm{addr}(t)$, then $s[t^c]$ wfa.

Proof. If $c \notin \mathrm{addr}(s)$ then $s[t^c] = s$ wfa by definition. If $c \in \mathrm{addr}(s) \setminus \mathrm{addr}(t)$ then each subterm $u^c \trianglerighteq s$ is changed to t^c but because of the condition $s@a = t@a$ for each a common to s and t no possible breach of the wfa condition can result. □

Because of this it is clear that it is reasonable to use updating as the basis for our extension to a 'sharing version' of any reduction over some inductively defined notion of terms. Notice that the thus defined sharing reduction is never contextual on the addressed terms, of course, even when it is based on a contextual reduction, because any reduction is essentially global in that all subterms with a particular address throughout the entire term must be reduced simultaneously. This is *parallel reduction* as found in rewriting. We will need this notion sufficiently often that it warrants employing a new notion of 'closure' that mimicks the contextual closure in a context with sharing.

3.1.5 Definition (sharing reduction). Assume a set of inductively defined T-terms and a reduction $\rightarrow$ on sharing T-terms. The *sharing extension* of $\rightarrow$, notation

$\mathrel{+\!\!\!\!\rightarrow}$, is the reduction on $\mathbb{T}$wfa defined as follows: Given $t \in \mathbb{T}$wfa which can be written as $t = C[r]$ such that $r \rightarrow p$. Now consider $C[r]$:

Case r is unshared in $C[r]$: then $C[r] \mathrel{+\!\!\!\!\rightarrow} C[p]$.

Case r is shared in $C[r]$: then it is possible to express $C[r]$ on the form $C_1[(C_2[r])^a]$ for some address a, where r is unshared in $C_2[r]$ (*i.e.*, $C_2[\,]$ contains no addresses) . Then

$$t \mathrel{+\!\!\!\!\rightarrow} \left(C_1[(C_2[p])^a]\right) \left[(C_2[p])^a\right]$$

where the update operation ensures that *all* the shared instances of r are contracted to p simultaneously.

By *sharing closure* we mean the sharing extension of the contextual closure.

We have used the notation $\mathrel{+\!\!\!\!\rightarrow}$ to suggest that this is *parallel reduction* in that all subterms labelled with some particular label are reduced 'simultaneously'. A simple consequence of this is the following:

3.1.6 Proposition. For all reductions $\rightarrow$, $\begin{tikzcd} \blacksquare \arrow[r, "\mathrel{+\!\!\!\!\rightarrow}"] \arrow[d, "\rhd"'] & \blacksquare \arrow[d, "\rhd"] \\ \blacksquare \arrow[r, dotted, "+"] & \blacksquare \end{tikzcd}$,where $\rhd\!\!\longrightarrow$ is the relation defined by $t \rhd\!\!\longrightarrow \triangle(t)$.

Proof. Assume $t \mathrel{+\!\!\!\!\rightarrow} s$. Then from Definition 3.1.5 we have $t = C[r]$ in both cases: in the unshared case the result is trivial; in the shared case it is clear from the wfa condition that $t@a = C_2[r]^a$ and hence $s@a = C_2[p]^a$ by the updating operation. Since wfa guarantees that there is no overlap between the subterms with address a, $\triangle(t) \xrightarrow{+} \triangle(s)$. $\square$

This guarantees the correctness of sharing reduction. However, it does not establish the soundness, and in fact this is nontrivial and must be established for each kind of rewrite system separately. The following example explains the problem for the TRS case; we will return to the situation for λ-graphs below.

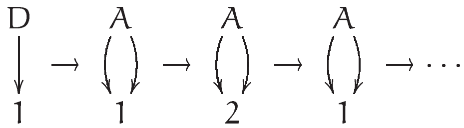
3.1.7 Comparison (term graph rewriting). In Barendregt et al. (1987) the notion of *graph reducible* is introduced which captures that a term graph rewriting system (TGRS) is sound and correct with respect to a term rewrite system (TRS) which is 'the same without sharing'. In fact, the main result of Barendregt et al. (1987, Theorem 5.12) is that every orthogonal TRS can be interpreted as a TGR system which implements the same reduction. Orthogonality is a rather severe restriction on the rules: Barendregt et al. (1987) quote the following counterexample to why it is necessary: the TRS

$$A(1,2) \to 0 \qquad 1 \to 2 \qquad 2 \to 1 \qquad D(x) \to A(x,x)$$

includes the reduction

$$D(1) \to A(1,1) \to A(1,2) \twoheadrightarrow 0$$

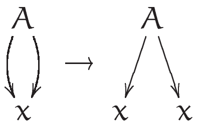
which is not possible with sharing: here the only reduction is



This exists equivalently for our sharing extension $\mathrel{\Vert\!\!\longrightarrow}$ of $\to$ where it is expressed as

$$D(1) \mathrel{\Vert\!\!\longrightarrow} A(1^a, 1^a) \mathrel{\Vert\!\!\longrightarrow} A(2^a, 2^a) \mathrel{\Vert\!\!\longrightarrow} A(1^a, 1^a) \mathrel{\Vert\!\!\longrightarrow} \cdots$$

The above problem can be solved by allowing arbitrary *copying*, of course: if the relation (in the TGR formulation) has the rule



added then the problem does not arise. The morale is this:

> " Graph rewriting faithfully models sharing but not the necessary copying which happens behind the scenes. "

With this insight we turn to λ-graph rewriting.

3.1.8 Definitions (λa-terms).

1. The set of λ*a-preterms* is the set of sharing λ-preterms.

2. The notion of *free variables* for λa-preterms is the one for λ-preterms (Definition A.3.2.1) generalised to any λa-preterm M by the equation

$$\mathrm{fv}(M^a) = \mathrm{fv}(M)$$

3. *Renaming* for λa-preterms is as for λ-preterms (Definition A.3.2.2) generalised to any λa-preterm M by

$$M^a[y := z] = M[y := z]^a$$

4. *α-equivalence* for λa-preterms is as for λ-preterms (Definition A.3.2.3) generalised to any λa-preterm M by the equation

$$M^a \equiv N^a \qquad \text{if } M \equiv N$$

(so $\equiv$ does not allow renaming of addresses).

5. The set of *λa-terms*, is Λa $= \Lambda$ wfa, is the λa-preterms modulo $\equiv$.

We will confuse preterms and terms as usual, employing the variable convention and assuming that all explicitly mentioned addresses are distinct; sometimes we will introduce *fresh* addresses by which we mean globally unique ones.

Here the problem that may occur when applying β-reduction naïvely to λ-graphs is that (parts of) the abstraction of the redex might be 'too shared' from a context that is different from the one of the redex. This creates a need for 'spontaneous copying'. Consider, for example, reduction of the λa-term above. If we reduce a shared redex it seems obvious that somehow the reduction must happen in all instances simultaneously. Here is one such β-reduction (the redex is underlined):

$$\begin{array}{c}
(\lambda x.wx(\lambda z.zv)^b)^a \ \underline{\big((\lambda x.wx(\lambda z.zv)^b)^a \ (\lambda z.zv)^b\big)} \\
\big\downarrow ? \\
(\lambda x.wx(\lambda z.zv)^b)^a \ ((\lambda z.zv)^b x(\lambda z.zv)^b)^a
\end{array}$$

which is no longer wfa: the a address no longer shares a unique subterm! The 'fix' of just forcing the other shared term to be the same,

$$\begin{array}{c}
(\lambda x.wx(\lambda z.zv)^b)^a \ \underline{\big((\lambda x.wx(\lambda z.zv)^b)^a \ (\lambda z.zv)^b\big)} \\
\big\downarrow ? \\
((\lambda z.zv)^b x(\lambda z.zv)^b)^a \ ((\lambda z.zv)^b x(\lambda z.zv)^b)^a
\end{array}$$

is plain wrong, of course.

The crucial insight of Wadsworth is that substitution should only happen in the *instance of the abstraction that participates in the redex*. In this case we should make sure that the abstraction $(\lambda x.wx(\lambda z.zv)^b)^a$ is 'unshared' sufficiently for the substitution of x in it to be safe. So we need to allow partial unraveling to get correct reduction. Here is one way of doing it by simply removing the a address completely (this correponds to copying the subterm at a into as many copies as there are instances):

$$\begin{array}{c}
\underline{(\lambda x.wx(\lambda z.zv)^b)^a}\ \big(\underline{(\lambda x.wx(\lambda z.zv)^b)^a}\ (\lambda z.zv)^b\big) \\
\Big\downarrow{\scriptstyle \mathrm{cp}} \\
(\lambda x.wx(\lambda z.zv)^b)\ \big((\lambda x.wx(\lambda z.zv)^b)\ (\lambda z.zv)^b\big) \\
\Big\downarrow \\
w\big((\lambda x.wx(\lambda z.zv)^b)\ (\lambda z.zv)^b\big)(\lambda z.zv)^b
\end{array}$$

where we have used $\xrightarrow[\mathrm{cp}]{}$ to denote copying, *i.e.*, removing addresses). In fact this turns out to be the least change we can do in this case: Wadsworth defined what is needed for a redex R as R-*admissible*, and proved that reducing a redex which is R-admissible will always yield an admissible graph. The definition of Wadsworth (which we return to below) is extremely simple: in our notation it just requires the abstraction subterm itself to not have an address. However, removing this address means that other adjustments may be needed to satisfy the general wfa property – it just happens that in the example above this was trivial. This is formalised as follows: first the restriction.

3.1.9 Definitions (admissibility). Given a λa-term $M \equiv C[R]$ where we intend to reduce the redex $R \equiv (\lambda x.P)Q$. M is R-*admissible* if M wfa and it is possible to find a λa-context $C'[]$ and subterms $P_1, \ldots, P_n \trianglerighteq P$ (for some $n \geq 0$) such that Let $R \equiv (\lambda x.P)Q$ be a redex. $M = C[R]$ is R-addmissible if

1. $P \equiv C'[P_1, \ldots, P_n]$,
2. For all the P_i, $x \notin \mathrm{fv}(P_i)$.
3. There is no sharing between $C'[]$ and the rest of M, *i.e.*,

$$\mathrm{addr}(C[\,]) \cap \mathrm{addr}(C'[\,]) = \varnothing$$

(M wfa ensures $(\mathrm{addr}(P_1) \cup \cdots \cup \mathrm{addr}(P_n)) \cap \mathrm{addr}(C'[\,]) = \varnothing$).

Furthermore, if M is R-admissible for all $R = (\lambda x.P)Q \trianglerighteq M$ then we say that M is $\xrightarrow[\beta]{}$-*admissible* or merely *admissible*.

Next the copy operation.

3.1.10 Definitions (copying). Define the following additional reductions.

1. *Copying* is the sharing closure of
$$M^a \to M \tag{cp}$$

2. $C[R]$ R-*copies* to N, notation $\xrightarrow[\text{R-cp}]{}$, iff $C[R] \underset{\text{cp}}{\twoheadrightarrow} C[R'] \equiv N$ and $C[R']$ is R'-admissible.

This notion of reduction is clearly well-defined thanks to Proposition 3.1.4 and Theorem 3.1.11. It is intentionally not deterministic: the variation can be used to choose the alternatives from full laziness to copying by doing the minimal and maximal number of $\underset{\text{cp}}{\to}$-steps.

This gives a result very similar to Wadsworth (1971, Theorem 4.3.15) but much easier to prove.

3.1.11 Theorem. *Given the* λa*-term* $M \equiv C[R]$ *with* $R \equiv (\lambda x.P)Q$ *and let* M *be* R*-admissible. Then* $C[R']$ *with* $R' \equiv P[x := Q]$ *is a* λa*-term.*

Proof. We know that $R \equiv (\lambda x.P)Q$ with $P \equiv C'[P_1, \ldots, P_n]$ from Definition 3.1.9. The reduction must establish $R' \equiv P[x := Q] \equiv C'[P_1, \ldots, P_n][x := Q] \equiv C'[x := Q][P_1, \ldots, P_n]$ since $x \in \mathrm{fv}(P_i)$. Since there are no addresses shared between $C'[]$ and the rest of M this substitution cannot ruin the wfa property. However, we still need to consider what happens *inside* $C'[x := Q][]$: here we may have a subterm N^a occurring in several places but substituting $N^a[x := Q]$ clearly also preserves the wfa property. □

Before we proceed with the definition of reduction, we relate to the original definitions of Wadsworth.

3.1.12 History (Wadsworth's lambda-graphs). The intuition of the above is to use 'wfa' to express that a preterm corresponds to a λ-term with properly defined sharing information. This follows the tradition of the λ-graph calculus of Wadsworth (1971, Chapter 4), where it is called *admissible* (more on this in History 3.1.13 below).

Wadsworth's formulation uses a different notation, namely pictures of the rooted directed graph obtained by drawing each abstraction as a *rator node* λx (for the appropriate x), each application as a *rand node* γ, each variable as a *terminal node* x (again for the appropriate x), and include a unique number

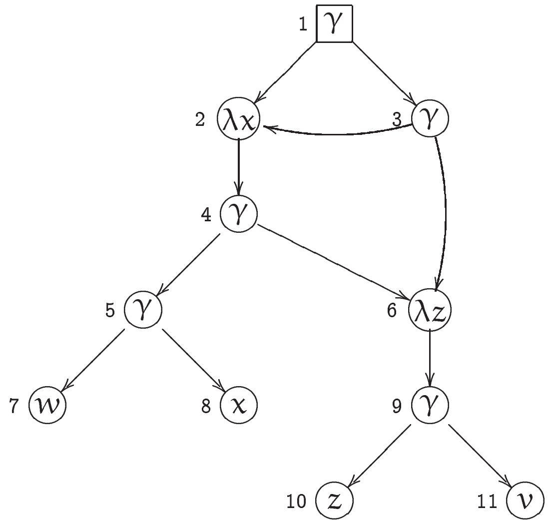


Figure 3.1: Example Wadsworth λ-graph.

for each node. Node identity is then used for sharing, *i.e.*, shared subterms *are* the same subgraph. For example, the λa-term

$$(\lambda x.wx(\lambda z.zv)^b)^a \ \big((\lambda x.wx(\lambda z.zv)^b)^a \ (\lambda z.zv)^b\big)$$

is depicted in this style in Figure 3.1 (mimicking Wadsworth 1971, Figure 5a) where ^{a} and ^{b} identify the nodes 2 and 6, respectively.

Furthermore notice that variables are syntactic in Wadsworth's graphs: instead of the variable convention Wadsworth explicitly requires all variables occuring in the graph to be unique, and Wadsworth allows 'constants' which we will not discuss here (but see the treatment of CRS in chapter 4). Except for this it can be seen that our λ-graph notion is very close to Wadsworth's.

3.1.13 History (Wadsworth's admissibility and full laziness). To be exact, Wadsworth (1971, pp.150-152) defines the following: A graph G is *admissible* if

(A1) G is acyclic and unravels to a proper λ-preterm.

(A2) Each terminal node has a unique binder.

(these demands correspond directly to wfa except for variables being considered syntactically). The graph is R-*admissible* if furthermore

(A3) The rator-node of R is accessible only via the rator-pointer of R.

The last demand is the same as our three demands because (A2) means that all occurrences of the particular variable x bound by the rator-node in question *must* be reachable through that rator-node; this means that the demand that the rator-node is not shared implies that none of the nodes corresponding to our $C'[\,]$ can be shared from the 'outside'.

In fact Wadsworth goes further and defines the notion *directly abstractable* that guarantees R-admissibility. This is especially interesting because in essence it subsumes the later notion of *maximal free subexpressions* to obtain *full laziness* (Hughes 1982, Johnsson 1984, Peyton Jones 1987)! In this respect Wadsworth's definition is akin to the author's *graph matching ordering* (Rose 1993) which is again the same as the *graph bisimilarity* of Ariola and Klop (1995a) in that it permits the full range from fully lazy reduction, when $C'[\,]$ is as small as possible, to simple λ-body copying, when $C'[\,]$ is as large as possible (*i.e.*, is P).

Finally we can define the shared β-reduction step using the techniques introduced above.

3.1.14 Definitions (λa-reduction). Define the following reductions on Λa (modulo $\equiv$).

1. R-*contraction* is the sharing closure of the (β) rule.
2. *Shared* β*-reduction* of $C[R]$, notation $\xrightarrow[\beta a]{}\!\!\!\!\!\!\!\!+\;\;\;$, is defined as R-copying followed by R-contraction.

This notion of reduction is clearly well-defined thanks to Proposition 3.1.4 and Theorem 3.1.11. It is intentionally not deterministic: the variation can be used to choose the alternatives from full laziness to copying by doing the minimal and maximal number of $\xrightarrow[cp]{}$-steps.

3.1.15 History. Wadsworth (1971, p.160) actually defines a $\mathrm{Copy}(R, G)$ operation concretely as doing complete λ-body copying, and uses the 'directly abstractable' notion to obtain full laziness.

The essential issue thus remains that addresses are global so the entire context must be considered continuously. The complexity of this makes reasoning about shared β-reduction somewhat difficult – in fact the history of sharing as we have referred to it throughout is a long path of how this is best described (or swept under the rug). This is usually safe because of the following result which is usually implicitly included.

3.1.16 Definition (sharing stable). A reduction $\rightarrow$ is *sharing stable* iff $\xrightarrow{+} \subseteq \mathrel{+\!\!\!+\!\!\!\rightarrow}$, *i.e.*, if the sharing extension is sound (the opposite of Proposition 3.1.6).

Several obvious examples exist of sharing stable reductions. We mention two.

3.1.17 Propositions (sharing stable). Given a confluent and contextual reduction, $\rightarrow$.

1. The sharing extension of $\underset{\mathrm{cp}}{\twoheadrightarrow\!\!\!\!\!\rightarrow} \cdot \mathrel{+\!\!\!+\!\!\!\longrightarrow}$ is sharing stable.

2. The sharing extension of the relation $\underset{\mathrm{cp}}{\rightarrow} \cdot \mathrel{+\!\!\!+\!\!\!\longrightarrow}$ restricted to contexts given by

$$\frac{\mathrm{M} \twoheadrightarrow \mathrm{C}[\mathrm{R}] \quad \wedge \quad \mathrm{R} \rightarrow \mathrm{P}}{\mathrm{M} \rightsquigarrow \mathrm{C}[\mathrm{P}]} \quad \text{if } \mathrm{R} \textit{ unshared in } \mathrm{C}[\mathrm{R}]$$

   is sharing stable.

Proofs. 1 is trivially true since the complete unraveling constructed by $\underset{\mathrm{cp}}{\twoheadrightarrow\!\!\!\!\!\rightarrow}$ is onbviously admissible for any reduction. 2 is equally trivial since 'unshared' implies admissibility in any context. □

Last we summarise some of the attempts at addressing this difficult problem in the literature – our attempt in the following section is based on the combination with explicit substitution for which it is easier and furthermore have nicer complexity properties.

3.1.18 Comparisons.

1. Felleisen and Friedman (1989, Section 4) use a syntax very close to what is presented above. However, the resulting calculus is rather more complicated. In particular the rules are all expressed over arbitrary contexts. It is interesting to note that our $\mathrel{+\!\!\!+\!\!\!\xrightarrow[\beta\mathrm{a}]{}}$ is explicitly not used by Felleisen and Friedman because their calculus would then perform a wrong subtitution of a bound variable. The problem is described as "establishing the sharing relation too early" – the same thing happens with $\mathrel{+\!\!\!+\!\!\!\xrightarrow[\beta\mathrm{a}]{}}$ if one insist on a strategy that breaks admissibility but we have the choice of not doing this.

2. Launchbury (1993) uses a compositional description maintaining a *store* component mapping all addresses to their contents. This means that the sharing technology is intimately tied with the chosen reduction strategy which is coded into the system (in this case lazy reduction). However, the well-formed addressing problem is not really solved compositionally in this approach either because there is a special $\hat{=}$ operator used to enforce the uniqueness of addresses in substitution bodies.

3. The acyclic component of λ-graph rewriting of Ariola and Klop (1994) is very close to what we have presented above. However their definition is not in any obvious way an extension of the λβ-calculus but rather based on the sharing inherent in *mutually recursive equations*. The problem is manifest in what the authors call the *scope cut-off*: the idea that sharing 'breaks' the scope of λ-abstraction. We have more to say for both cases when we treat cyclic data in section 4.3.

3.2 Explicit Substitution & Sharing

In this section we demonstrate how explicit sharing can be added to the λxgc-calculus in such a way that we avoid the problems encountered when trying to add sharing 'first', as discussed in the previous section.

The advantage of doing this is that a calculus is obtained where all the copying performed implicitly in the λ-graph case is done explicitly. In particular, the optional garbage collection facility will give a quantification of full laziness.

We will first introduce the synthesis of λa and λxgc, including the notion of reduction.

3.2.1 Definition (λxa-terms). The set of λxa-preterms is ranged over by the letters $MN\ldots$ (*etc*, as usual), and defined inductively by

$$M ::= x \mid \lambda x.M \mid MN \mid M\langle x := N\rangle \mid M^a$$

where abc range over an infinite set of *addresses*. The new form M^a is called an *addressing* where a *addresses* M (as for λa-terms). The wfa and unraveling definitions are lifted from Definition 3.1.1; the notions of free variables, renaming, and α-equivalence, are defined using the combination of the rules for λa- and λx-terms in the obvious way. As usual the λxa-*terms* Λxa, are the wfa preterms modulo $\equiv$.

So what should a sharing reduction step look like with explicit substitution? The obvious generalisation of λxgc-reduction is to ensure that any *duplication* in the rules is remembered in the form of a *sharing introduction* – this ensures that a realistic implementation can avoid the *copying overhead*. Now all we have to show is that the resulting reduction has sharing stable reduction.

Fortunately, a quick glance over the λxgc-rules reveals that the only thing that is ever duplicated is a substitution (namely by (xap)). Thus *we only have to share substitutions*. Furthermore, we need to ensure that addresses do not 'block' redexes by being squeezed in the middle of a β- or x-redex. Collecting this insight we get the following system, our starting point.

3.2.2 Definition (λxgca-reduction step). $\xrightarrow[\text{bxgcacp}]{}$ is the contextual closure of

$$\begin{aligned}
(\lambda x.M)N &\to M\langle x := N\rangle^{a} \quad a \text{ fresh} && \text{(ba)}\\
x\langle x := N\rangle &\to N && \text{(xv)}\\
y\langle x := N\rangle &\to y \quad x \neq y && \text{(xvgc)}\\
(\lambda y.M)\langle x := N\rangle &\to \lambda y.M\langle x := N\rangle && \text{(xab)}\\
(M_1 M_2)\langle x := N\rangle &\to M_1\langle x := N\rangle\ M_2\langle x := N\rangle && \text{(xap)}\\
M\langle x := N\rangle &\to M \quad x \notin \mathrm{fv}(M) && \text{(xgc)}\\
M^{a} N &\to MN && \text{(cpap)}\\
M^{a}\langle x := N\rangle &\to M\langle x := N\rangle && \text{(cpx)}
\end{aligned}$$

We write $\xrightarrow[\text{bxgca}]{}$ for the relation without (cpap,cpx); $\xrightarrow[\text{bxgca}]{\#}$ is the sharing closure of $\xrightarrow[\text{bxgcacp}]{}$.

The interesting thing about this system is that all copying happens explicitly: before any reduction we need at most unravel 'one level' of sharing before we can reduce the redex safely.

Formally, this is because 'admissibility' of Definition 3.1.9 is really a notion of the *redex* involved in a rewrite rather than of the *term*. Since our reductions above do not perform substitution, wfa is maintained everywhere outside the small affected area of the term. Inspired by this we define a restricted form of admissibility applicable for λxgca.

3.2.3 Definitions (local admissibility). Given an λxa-term $M \equiv C[C_R[\vec{P}]]$ where we intend to reduce the redex $C_R[\vec{P}]$ with a rule $C_R[\vec{v}] \to C'_R[\ldots]$. M is *locally*

R-*admissible* if there is no sharing between $C_R[\,]$ and the rest of M, *i.e.*,

$$\mathrm{addr}(C[\,]) \cap \mathrm{addr}(C'[\,]) = \varnothing$$

(M wfa ensures $(\mathrm{addr}(P_1) \cup \cdots \cup \mathrm{addr}(P_n)) \cap \mathrm{addr}(C'[\,]) = \varnothing$). Furthermore, if M is R-admissible for all $N \equiv C_R[\vec{P}] \trianglerighteq M$ for each rule R then we say that M is *locally admissible* with respect to the reduction, and a reduction that preserves local admissibility everywhere is called *locally sharing stable*.

This notion of local admissibility is sufficient for safely reducing even with sharing when all changes are local as for the λxgca-calculus; notice that this, like Proposition 3.1.17, is the opposite of the (also here inherent) Proposition 3.1.6.

3.2.4 Lemma. If $\mathrel{\mapstochar\twoheadrightarrow}$ is locally sharing stable for λxgca-reduction then it satisfies for any λx-term M,

$$\begin{tikzcd} \square \arrow[r, dotted, "+", "\mapstochar"'] \arrow[d, dotted, two heads, "\mathrm{cp}"'] & \square \arrow[d, dotted, two heads, "\mathrm{cp}"] \\ M \arrow[r, "\mathrm{bxgc}"'] & \blacksquare \end{tikzcd}.$$

Proof. This is easy by observing that no matter how evil an addressing we introduce by the 'cp-expansion' allowed (going 'up' from M in the diagram), this will not introduce any sharing that prevents us from reaching some term that unravels back to the λxgc-reduct because any λxgc-redex will be reestablishable as a corresponding λxgca-redex. □

3.2.5 Theorem. $\underset{\mathrm{cp}}{\twoheadrightarrow}$ *is a projection from* $\xrightarrow[\mathrm{bxgca}]{+}$ *to* $\xrightarrow[\mathrm{bxgc}]{+}$

Proof. One way is Proposition 3.1.6, the other Lemma 3.2.4. □

Notice how this has as a consequence that *reduction with sharing is at least as efficient as without* (for any reduction strategy) if we disregard the copy-operations. The following are obvious consequences.

3.2.6 Corollary. $\xrightarrow[\mathrm{bxgca}]{}$ *is a conservative extension that preserves strong normalisation of* $\xrightarrow[\beta]{}$.

Proof. Combination of Theorem 3.2.5 and Theorem 1.1.17. □

We have now got rid of the problem of the size of the abstraction body and how much to copy, a gain we would expect given that all redcutions are local to the part of terms which is matched. The system *still* covers the full range of

Syntactic Domains. The λ_{let}-terms are define by

$$
\begin{aligned}
V, W &::= \lambda x.M && \text{(Values)}\\
A &::= V \mid \mathsf{let}\ x = M\ \mathsf{in}\ A && \text{(Answers)}\\
L, M, N &::= x \mid M\ N \mid \mathsf{let}\ x = M\ \mathsf{in}\ N && \text{(Terms)}
\end{aligned}
$$

where xyz range over variables.

Axioms. Reduction, $\xrightarrow[need]{}$, is the contextual closure of

$$
\begin{aligned}
(\lambda x.M)N &\to \mathsf{let}\ x = N\ \mathsf{in}\ M && \text{(let-I)}\\
\mathsf{let}\ x = \lambda y.V\ \mathsf{in}\ C(x) &\to \mathsf{let}\ x = \lambda y.V\ \mathsf{in}\ C(\lambda y.V) && \text{(let-V)}\\
(\mathsf{let}\ x = L\ \mathsf{in}\ M)\ N &\to \mathsf{let}\ x = L\ \mathsf{in}\ M\ N && \text{(let-C)}\\
\mathsf{let}\ y = (\mathsf{let}\ x = L\ \mathsf{in}\ M)\ \mathsf{in}\ N &\to \mathsf{let}\ x = L\ \mathsf{in}\ \mathsf{let}\ y = M\ \mathsf{in}\ N && \text{(let-A)}
\end{aligned}
$$

Figure 3.2: The Call-by-Need calculus, λ_{let}.

possible reduction strategies and from fully lazy to complete copying. The choice is in (gc) because we *eliminate addresses when distributing substitutions*: if we apply (gc) at some point because it is possible then we avoid distributing subsitutions – copying – underneath that point. This is what took so much effort with directly abstractable subterms/maximal free subexpressions in the general substitution case.

3.2.7 Comparison (Call-by-Need). An interesting comparison is with the 'standard reduction' of λ_{let}-calculus of Ariola, Felleisen, Maraist, Odersky and Wadler (1995, Figure 2), reproduced in Figure 3.2. This implements the *Call-by-Need* reduction strategy which is weak reduction with sharing of unevaluated terms.

This is close to λxgca with a *Call-by-Value restriction on copying* similarly to what we would get by restricting (cpx) to

$$
M^{a}\langle x := N\rangle \to M\langle x := N\rangle \quad \text{if } M \text{ is a value} \qquad (\mathrm{cpx}_V)
$$

This means that *sharing interferes with scoping* as also discussed by the authors using an elaborate system of nested boxes. The main problem with this, compared to our approach, is that *copying changes scoping* – in fact sharing variables are indistinguishable from λ-bound variables in λ_{let} (this can be seen as an advantage, of course). This is not a problem in our calculus. The price we pay is that sharing remains a global concern. This can be remedied through use of appropriate strategies, as we shall see in chapter 4. And in fact the problem is also present implicitly in λ_{let} through the fact that the rule (let-V) allows 'instant substitution' into any context.

Another interesting aspect of λ_{let} is that it has *substitution of composition* in the (let-A) rule and hence is not likely to PSN of $\lambda\beta$-reduction as discussed in section 1.2.

3.2.8 Discussion (complexity). We have argued throughout that combination of explicit substitution and sharing gives a representation of the λ-calculus which is faithful to the semantics on one side and the complexity of mechanical evaluation on the other – this was, in fact, the chief motivation for environment-based evaluation and Call-by-Need (Henderson 1980). The crucial argument is, in fact, the notion of *locality* used above: computers are good at shuffling data around within a bounded fragment of the world, *cf.* the discussion in the introduction. While it may not be a universal fact that data must be accessed through *addresses* it is certain that some notion of *space* and consequenctly *distance* will exist in computers, hence locality will always be important. And this is the main issue that we have obtained above: both the issues of *substitution* and *copying* have been made local in the rewriting system. Hence they can be realised by 'shufflng' on a computer which means they can be implemented efficiently. It is also worth noting that *using explicit naming does not change the complexity measures in any asymptotic way* as argued in section 2.2.

3.2.9 Discussion (sharing vs de Bruijn indices). As discussed in section 2.1, the original definition of de Bruijn (1972) used three concepts for the implementation of substitution (shown in Definition 2.1.1), and the various naming principles can seen as combinations of these. This of course raises the issue as to how the different naming principles interfere with, in particular, explicit sharing. This is connected to the way de Bruijn indices are usually seen as a description of an abstract 'stack'. While this is a nice idea it *is sensitive to the reduction history* because different reduction sequences yield different configurations of the chosen shift/lift/*etc* operators as witnessed by the comparison in Figure 2.2 where the two $\lambda\upsilon$-terms $\lambda\lambda(123)[\Uparrow(1/)]$ and $\lambda\lambda(213)[2/]$ are identical but have no common reduct short of the substitution normal form. Furthermore the fact that no complexity information is improved as argued above means that we will not study the combination of sharing and explicit naming such as de Bruijn indices.

3.3 Discussion

In this and the two previous chapters we studied *syntactic extensions* in that each extension makes some operational aspect of reduction explicit. Specifically, explicit sharing, explicit binding, and explicit substitution can be defined in such a way that the highest degree of orthogonality between the concepts is obtained. We show this by demonstrating how the large number of published extensions of λ-calculi can be understood as 'points in λ-space'. The purpose of the separation is to reason about classes of extensions by using projections that preserve the property, *i.e.*, ignore as many 'irrelevant' dimensions as possible.

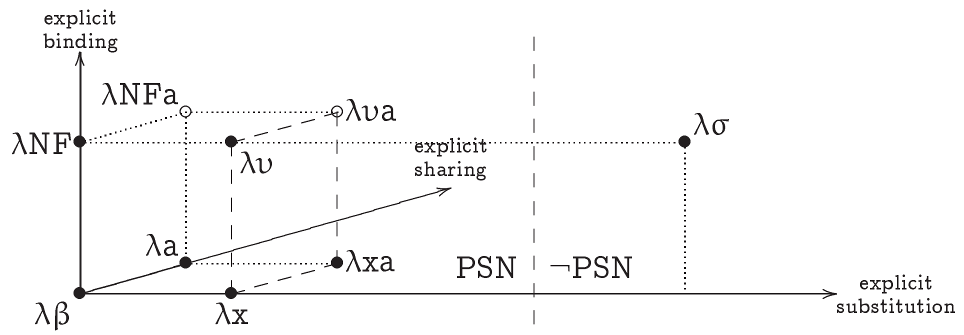


Figure 3.3: Explicitness dimensions and selected calculi.

The situation is illustrated in Figure 3.3: the origin is the (untyped) λβ-calculus, and we give samples of a 'unit' of each dimension (where the dimension is first fully explicit): λNF is de Bruijn's (1972) 'namefree' calculus described in section 2.1; we will say that this has *explicit binding*. λx is our calculus with *explicit substitution* in the most abstract way. Finally, λa is the variation of Wadsworth's (1971) λ-graph calculus using (abstract) *addresses* of section 3.1 which we will say has *explicit sharing*.

The combination of explicit substitution and sharing into a calculus modelling the *complexity of sharing independent of the reduction strategy* is new in an abstract calculus: all published systems known to us either restrict reduction (Ariola et al. 1995, Crégut 1990, Felleisen and Friedman 1989, Grue 1987, Hughes 1982, Staples 1978, Sestoft 1994, Turner 1979, Toyama, Smetsers, van Eekelen and Plasmeijer 1991), or use 'global' graph (or 'store') structure (Ariola and Klop 1994, Barendregt et al. 1987, Burn, Peyton Jones and Robson 1988, Benaissa and Lescanne 1996, Glauert, Kennaway and Sleep 1989, Jeffrey 1993, Koopman, Smetsers, van Eekelen and Plasmeijer 1991, Launchbury 1993, Plotkin 1981, Rose 1996, Staples 1980), if it is not hidden as was the case in the earliest descriptions; our solution is essentially to use an *address oracle* which *cannot be observed* except for the sharing relation it represents. This means that we *isolate the complexity issue* for λ-calculus implementations – an important first step towards solving the problem for functional languages. As a side contribution we observe that λxa is β-PSN which guarantees that our quest for 'realistic' computation times has not introduced the possibility of infinite reductions.

Chapter 4

Higher-Order Rewriting and Functional Programs

In this chapter we demonstrate how some of the techniques of the previous chapters can be generalised to Klop's (1980) *combinatory reduction systems* (CRS, explained in section 4.1) to define a class of reduction systems that can be used to model a large class of aspects of evaluation of functional programming languages in a way that gives operationally faithful measures of time and space consumption by the paradigm

$$\begin{aligned} \text{reduction length} &= \text{time} \\ \text{term size} &= \text{space} \end{aligned}$$

The main part of the chapter constrains and extends CRSs to a class that can be executed efficiently on real computers: first, in section 4.2, we generalise *explicit substitution* to CRSs and augment with constraints that ensure CRS reduction to be not just *local* but also *efficient*, and we describe how such a system can be obtained automatically from any CRS. Second, in section 4.3, we generalise *sharing* to CRSs and it turns out this combination will also permit special treatment of *explicit recursion*. Finally, we discuss in section 4.4 how the class realises operational aspects of an actual functional programming language, namely Plotkin's (1977) PCF with sharing.

4.1 Combinatory Reduction Systems

Combinatory reduction systems (CRSs) form an expressive class of reduction systems over inductively defined terms extended with the variable binding notion of λ-calculus, by formalizing precisely the intuitive notion of substitution without extra constraints such as typing, algebraic conventions, *etc.* We study CRSs in this tutorial because they constitute a generalisation of the rewriting done by functional programming 'evaluation steps.' An *implementation* of CRS reduction is provided by Rose (1996b, chapter 6).

The CRS formalism was invented by Klop (1980) to provide a systematic treatment of combinations of term rewrite systems (TRS) with the λ-calculus, inspired by the 'definable extensions of the λ-calculus' of Aczel (1978). However, CRSs can also be understood as a form of 'higher order rewriting' (Nipkow 1991, van Oostrom and van Raamsdonk 1995).

This section gives a complete introduction to CRSs. The presentation is based on the (highly recommended) survey of Klop, van Oostrom and van Raamsdonk (1993), with some extensions and a slight change of notation to facilitate implementation and induction over syntax. Last we give two large example CRSs including sample reductions in full detail (created with the implementation of Rose (1996b, chapter 6)).

4.1.1 Definitions (combinatory reduction systems).

1. An *alphabet* is a set of *function symbols* $A, B, C, F, G, \ldots$ (and sometimes Greek letters and exotic symbols);each annotated with a *fixed arity* (notation F^n).

2. The corresponding *preterms*, denoted $t, s, \ldots$, have the inductive form

$$t ::= x \mid [x]t \mid F^n(t_1, \ldots, t_n) \mid Z^n(t_1, \ldots, t_n)$$

   where $x, y, z, \ldots$ are used to denote *variables* and X, Y, Z to denote *metavariables*, each with an explicit arity superscript as indicated.[1] The four different preterm forms are called *variable*, *metaabstraction*, *construction*, and *metaapplication* of a metavariable, respectively.

3. We denote the *set of metavariables* occuring in a preterm t as $\mathrm{mv}(t)$.

4. A preterm is *closed* if $\mathrm{fv}(t) = \varnothing$ where $\mathrm{fv}(x) = \{x\}$, $\mathrm{fv}([x]t) = \mathrm{fv}(t) \setminus \{x\}$, and $\mathrm{fv}(F^n(t_1, \ldots, t_n)) = \mathrm{fv}(Z^n(t_1, \ldots, t_n)) = \mathrm{fv}(t_1) \cup \cdots \cup \mathrm{fv}(t_n)$.

5. Metaterms are closed preterms considered *modulo renaming of (bound) variables*: we take as implicit the usual α-equivalence denoted ≡ and

[1] Thus for each particular alphabet the definition of preterms is finite and inductive.

defined inductively by $x \equiv x$, $[x]t \equiv [y]s$ if $t[x := z] \equiv s[y := z]$ for $z \notin \mathrm{fv}(s) \cup \mathrm{fv}(t)$, and $F^n(s_1, \dots, s_n) \equiv F^n(t_1, \dots, t_n)$ as well as $Z^n(s_1, \dots, s_n) \equiv Z^n(t_1, \dots, t_n)$ if $s_i \equiv t_i$ for each i; the renaming $t[y := z]$ is defined the usual way by $x[y := z] = z$ if $x = y$, $x[y := z] = x$ if $x \neq y$, $([x]t)[y := z] = [x']t[x := x'][y := z]$ with $x' \notin \mathrm{fv}([x]t) \cup \{y, z\}$, $F^n(t_1, \dots, t_n)[y := z] = F^n(t_1[y := z], \dots, t_n[y := z])$, and $Z^n(t_1, \dots, t_n)[y := z] = Z^n(t_1[y := z], \dots, t_n[y := z])$.

6. A *terms* is a metaterm without metavariables (but possibly including metaabstraction which is then called *abstraction*).

7. *Rewrite rules*, written $p \to r$, have the following restrictions:

   - the LHS (left hand side) p must be a *pattern*: it should be a construction, *i.e.*, the root of p is a function symbol, furthermore p should be closed and arguments of metaapplications in p should be distinct variables.
   - the RHS (right hand side) r must be a *contractor* of the LHS by which we mean that $\mathrm{mv}(p) \supseteq \mathrm{mv}(r)$.

8. A CRS is an alphabet together with a set of rewrite rules.

4.1.2 Notation (syntactic restrictions). We will sometimes define the preterms of a CRS by an inductive definition such as

$$a \ ::= \ x \ \mid \ L(a) \ \mid \ A(B(a_1), [x]a_2)$$

From such a definition it is easily seen that the alphabet of the defined CRS is $\{L^1, A^2, B^1\}$. However, more importantly, the declaration restricts the free term formation such that we only need to use the specified forms in inductive arguments, *e.g.*, it can be seen that B will only occur as the first component of an A-construction. We will use such *restricted (pre/meta)-terms* freely when it is clear that the defined *restricted CRS* subsystem is closed with respect to reduction.

4.1.3 Remark (term rewriting systems). A CRS restricted to not allow metaabstraction at all is a *term rewrite system* (TRS, also known as *first order rewriting systems*). TRS rules consequently have only 0-ary metavariables (in TRS literature these are just called "variables" since there are no bound variables in the CRS sense).

4.1.4 Example (λ-calculus). The ordinary untyped $\lambda\beta$-calculus is described by the CRS with alphabet $\{\lambda^1, @^2\}$ restricted by

$$t \ ::= \ x \ \mid \ \lambda([x]t) \ \mid \ @(t_1, t_2)$$

(to get rid of terms like $[x]x$ and $[x](\lambda x)$), and the single rule

$$@^2(\lambda^1([x]Z^1(x)), Y^0()) \to Z^1(Y^0()) \tag{β}$$

One can verify easily that (β) will reduce a restricted term to another restricted term.

The term $Z^1(Y^0())$ on the righthand side of (β) corresponds to the usual definition by substitution in that the bound variable to be substituted, x, is represented implicitly on the right side through the use of the metavariable Z^1 – informally we could have written $Z^1(Y^0())$ with ordinary substitution as something like $(Z^1(x))[x := Y^0()]$.

So the λ-term $(\lambda x.yx)y$ corresponds to the CRS term $@(\lambda([x]@(y, x)), y)$ and reduces by (β) to $@(y, y)$ if $Z^1(x)$ is matched with $@(y, x)$.

4.1.5 Notation (CRS abbreviations). We will use the usual CRS abbreviations:

- the arity superscript of function symbols and metavariables is omitted when obvious, and we omit the $()$ after zero-ary symbols,
- $[x, y, z]t$ abbreviates the nested metaabstraction $[x][y][z]t$,
- $Fxyz.t$ abbreviates 'curried F-abstraction' $F([x](F([y](F([z]t)))))$,
- st abbreviates 'application' $@(s, t)$ (when it has no other interpretation) where $@^2$ is then included as a function symbol, and
- $\vec{x}_{(n)}$, $\vec{Z}_{(n)}$, and $\vec{t}_{(n)}$, abbreviate the sequences $x_1, \ldots, x_n$, $Z_1, \ldots, Z_n$, and $t_1, \ldots, t_n$, respectively (we omit the $\cdot_{(n)}$ subscript when redundant).

We use $()$ to disambiguate where needed, and let application associate to the left and bind closer than abstraction – this allows ordinary conventions of rewriting and λ-calculus to be followed, *e.g.*, $\lambda xyz.xz(yz)$ denotes the rather unwieldy $\lambda^1([x](\lambda^1([y](\lambda^1([z]@^2(@^2(x, z), @^2(y, z)))))))$.

4.1.6 Example (λ-calculus, readable CRS). Using the above abbreviations we can express the λβ-calculus in a more usual notation: the following CRS is exactly the same as the one in Example 4.1.4 above:

$$(\lambda x.Z(x))Y \to Z(Y) \tag{β}$$

4.1.7 Example (Combinatory Logic). The class of *applicative TRS*, that is TRS where the only function symbol with non-zero arity is $@^2$, is naturally expressed using the above abbreviation. An example is combinatory logic:

$$\mathrm{SXYZ} \to (\mathrm{XZ})(\mathrm{YZ}) \tag{S}$$

$$\mathrm{KXY} \to \mathrm{X} \tag{K}$$

(but we should not write the RHS of (S) as $XZ(YZ)$ because that can be interpreted as $@(X^0, Z^2(@(Y^0, Z^0)))$ instead of the intended $@(@(X^0, Z^0), @(Y^0, Z^0))$.)

The substitution concept of CRS is reminiscent of a two-level λ-calculus in that a metavariable is always applied to the list of terms that should be substituted into its body. Metavariables are therefore instantiated to 'substitute-abstractions' denoted $\underline{\lambda}\vec{x}.t$ and the resulting 'substitute-redexes' play the rôle of substitution.

4.1.8 Definition (substitution). A *valuation* σ is a map that maps each metavariable Z^n to a *substitute* $\underline{\lambda}(\vec{x}_{(n)}).t$ where the x_i are distinct and t is a preterm. Valuations are homomorphically extended to metaterms: $\sigma(t)$ denotes the result of first inserting $\sigma(Z)$ for each metavariable Z in t and then replacing all the resulting *substitution* $(\underline{\lambda}(\vec{x}_{(n)}).t)(\vec{t}_{(n)})$ by $t[x_1 := t_1, \ldots, x_n := t_n]$ defined inductively by

$$\begin{aligned}
x_i[x_1 := t_1, \ldots, x_n := t_n] &\equiv t_i \\
y[x_1 := t_1, \ldots, x_n := t_n] &\equiv y \qquad y \notin \{\vec{x}\} \\
([y]t)[x_1 := t_1, \ldots, x_n := t_n] &\equiv [y](t[x_1 := t_1, \ldots, x_n := t_n]) \qquad y \notin \{\vec{x}\} \\
([x_i]t)[x_1 := t_1, \ldots, x_n := t_n] &\equiv \\
&[x_i](t[x_1 := t_1, \ldots, x_{i-1} := t_{i-1}, x_{i+1} := t_{i+1}, \ldots, x_n := t_n]) \\
F^m(\vec{s})[x_1 := t_1, \ldots, x_n := t_n] &\equiv \\
&F^m(s_1[x_1 := t_1, \ldots, x_n := t_n], \ldots, s_m[x_1 := t_1, \ldots, x_n := t_n]) \\
Z^m(\vec{s})[x_1 := t_1, \ldots, x_n := t_n] &\equiv \\
&Z^m(s_1[x_1 := t_1, \ldots, x_n := t_n], \ldots, s_m[x_1 := t_1, \ldots, x_n := t_n])
\end{aligned}$$

(the last case exists because one metavariable may be mapped to another when we allow reduction of metaterms). We say that s *matches* t when there exists a valuation σ such that $\sigma(s) \equiv t$.

A lot of complexity is hidden in the requirement that valuations be 'homomorphically extended' to metaterms because of the risk of name clashes. This is solved by the following definition which turns out to be a sufficient restriction for avoiding name conflicts.

4.1.9 Definitions (safeness).

1. The *bound variables* of a preterm, $\mathrm{bv}(t)$, are defined inductively as $\mathrm{bv}(x) = \varnothing$, $\mathrm{bv}([x]t) = \{x\} \cup \mathrm{bv}(t)$, $\mathrm{bv}(F^n(\vec{t}_{(n)})) = \mathrm{bv}(Z^n(\vec{t}_{(n)})) = \mathrm{bv}(t_1) \cup \cdots \cup \mathrm{bv}(t_n)$.

2. The rewrite rule $p \to t$ is *safe for the valuation* σ if for p and t considered as preterms

$$\forall Z \in \mathrm{mv}(p)\ \forall x \in \mathrm{fv}(\sigma(Z)) : x \notin (\mathrm{bv}(p) \cup \mathrm{bv}(t))$$

3. The valuation σ is *safe with respect to itself* if

$$\forall Z, Z' : \mathrm{fv}(\sigma(Z)) \cap \mathrm{bv}(\sigma(Z')) = \varnothing$$

Thus a CRS inherits the implicit complexity of the λ-calculus renaming, and resolves it in the same way by a generalised form of *variable convention*:

4.1.10 Convention (variable convention). Any valuation used is assumed safe with respect to itself, and any rule it will be used with is assumed safe with respect to it.

Clearly a safe variant can be obtained for any valuation in any context by renaming of some bound variables. This renaming is harmless since we consider metaterms modulo renaming.

4.1.11 Remark. It may seem contradictory that we discuss free variables at all considering that all metaterms are closed. This is because we may 'reduce under binding': a redex occurrence can easily be rooted inside an abstraction and thus have locally free variables. This suggests two interesting subcases:

- *TRS*s, defined above, do not include metaabstraction, and hence have no variables, so valuations of even contextual TRSs are automatically safe, and

- *weak* reduction (where all redexes are closed terms) only produces safe valuations.

This explains why abstract machines (TRS) and environment calculi (weak) avoid problems with valuation safety (for a treatment of these issues see Rose 1996b).

4.1.12 Definitions (match). Given a rule $p \to r$, some valuation σ (assumed safe in all contexts below, of course) defined such that $\sigma(p)$ is a term, and some context $C[\,]$. Then

1. $C[\sigma(p)]$ is *reducible*, $\sigma(p)$ is the *redex*, and $C[\,]$ is its *occurrence*.
2. $C[\sigma(r)]$ is the *contractum* of the rule, and
3. the pair $C[\sigma(p)] \to C[\sigma(r)]$ is a *rewrite* of the rule.

(in all cases we add "with match valuation σ" if appropriate). If $\sigma(p)$ is allowed to be a *metaterm* then we correspondingly define *metaredex*, *metacontractum*, and *metarewrite*, in fact we will use *meta*X to designate the property X of metarewriting.

4.1.13 Definition (reduction). Given a CRS R. R-*reduction* is the relation $\xrightarrow[R]{}$ generated by all rewrites possible with rewrites using rules from R. R-*metareduction* is the relation containing all metarewrites of the rules in R.

In this tutorial we are mostly concerned with confluent reduction systems. Confluence is, of course, in general undecidable, however, the *orthogonality* property defined by Klop (1980), identifies a decidable class of confluent CRSs. Here is a compact definition adapted from Klop et al. (1993):

4.1.14 Definitions (orthogonality). Let R be a CRS which consist of rules $p_i \to \ldots$.

1. R is *left-linear* if there are no repeated metavariables in any of the p_i.
2. All p_i must have the form $C[Z_1^{a_1}(\vec{x}_{1(a_1)}), \ldots, Z_n^{a_n}(\vec{x}_{n(a_n)})]$ where all metavariables are mentioned explicitly (hence the context $C[\,]$ is known to contain no metavariables in addition to being a construction). R is *nonoverlapping* if for any redex $\sigma(p)$ each other redex contained within it is also contained within some $\sigma(Z_j^{a_j}(\vec{x}_{j(a_j)}))$, *i.e.*, no constructor can be matched by more than one redex.

3. R is *orthogonal* if it is left-linear and nonoverlapping (some authors call this *regular*).

4.1.15 Theorem (Klop 80). *All orthogonal CRSs are metaconfluent.*

Proof. Confluence is proved in Klop et al. (1993, Corollary 13.6) using the notion of *parallel reduction*, $\twoheadrightarrow\!\!\!\!\!\!\;\shortparallel$, which contracts all (nonoverlapping) redexes simultaneously; metaconfluence is an easy consequence of the observation that the metavariables of a metaterm can be treated as otherwise not occurring function symbols. $\square$

This result can, in fact, be strengthened to *weakly orthogonal*[2] CRSs which means that overlaps are allowed when reduction of the two redexes always give the same result (proved by van Oostrom and van Raamsdonk 1995), however, we will not make use of this in this tutorial.

Finally, we an examples of a larger CRS. It is a generalisation of a system of Knuth (1973, p.337) exploiting higher-order rewriting.

4.1.16 Example (symbolic differentiation). Let the *arithmetic functions* (of one argument), ranged over by F, and the *arithmetic expressions*, ranged over by A and B, be defined inductively by

$$
\begin{aligned}
F &::= \lambda x.A \mid \ln \mid \exp \mid \cdots \\
A &::= x \mid FA \mid A+B \mid A-B \mid A\times B \mid \frac{A}{B} \mid i
\end{aligned}
$$

where x denotes any variable and i any integer constants. The *differential* dF of an arithmetic function F is derived by the CRS rules in Figure 4.1 (based on the list in Spiegel (1968, section 13)). One can use this to differentiate simple functions such as $\ln(x+1)$:

$$
\begin{aligned}
&\underset{\text{(function)}}{\boxed{d(\lambda x.(\ln)(x+1))}} \longrightarrow \lambda x.\underset{\text{(chain)}}{\boxed{D([x](\ln)(x+1),x)}} \\
&\longrightarrow \lambda x.\left(\underset{\text{(ln)}}{\boxed{d(\ln)}}\right)(x+1) \times \underset{\text{(plus)}}{\boxed{(D([x]x+1,x))}}
\end{aligned}
$$

[2] While orthogonal is the same as regular, *weakly regular* as defined by Barendregt, van Eekelen, Glauert, Kennaway, Plasmeijer and Sleep (1987) is a weaker notion where the reductions are merely required to commute.

First $d(f)$ that designates the differential of the function f.

$$d(\lambda x.X(x)) \longrightarrow \lambda x.D([x]X(x), x) \qquad \text{(function)}$$

Next the differentials of some standard functions.

$$d(\ln) \longrightarrow \lambda x.\frac{1}{x} \quad (\ln) \;\;;\;\; d(\exp) \longrightarrow \exp \quad (\exp)$$

Differentiating an expression e with respect to a variable x to be substituted by s is designated $D([x]e, s)$ (Knuth used $D_x(e)$ corresponding to $D([x]e, x)$ because the first-order system only permits trivial substitution).

$$D([x]Y, Z) \longrightarrow 0 \qquad \text{(constant)}$$

$$D([x]x, Z) \longrightarrow 1 \qquad \text{(identity)}$$

$$D([x](U(x) + V(x)), Z) \longrightarrow D([x]U(x), Z) + D([x]V(x), Z) \qquad \text{(plus)}$$

$$D([x](U(x) - V(x)), Z) \longrightarrow D([x]U(x), Z) - D([x]V(x), Z) \qquad \text{(minus)}$$

$$D([x](U(x) \times V(x)), Z) \longrightarrow (D([x]U(x), Z) \times V(Z)) + (U(Z) \times D([x]V(x), Z)) \qquad \text{(times)}$$

$$D([x]\frac{U(x)}{V(x)}, Z) \longrightarrow \frac{(D([x]U(x), Z) \times V(Z)) - (U(Z) \times D([x]V(x), Z))}{V(Z) \times V(Z)} \qquad \text{(divide)}$$

The interaction between functions is described as follows.

$$D([x](U)(V(x)), Z) \longrightarrow (d(U))(V(Z)) \times D([x]V(x), Z) \qquad \text{(chain)}$$

Finally applications need to be contracted.

$$(\lambda x.X(x))Y \longrightarrow X(Y) \qquad (\beta)$$

A few algebraic simplifications ... (many more can be added, of course).

$$0 + X \longrightarrow X \quad (0{+}x) \;\;;\;\; X + 0 \longrightarrow X \quad (x{+}0) \;\;;\;\; X - 0 \longrightarrow X \quad (x{-}0)$$

$$1 \times X \longrightarrow X \quad (1x) \;\;;\;\; X \times 1 \longrightarrow X \quad (x1) \;\;;\;\; 0 \times X \longrightarrow 0 \quad (0x) \;\;;\;\; X \times 0 \longrightarrow 0 \quad (x0)$$

$$\frac{X}{1} \longrightarrow X \quad (x/1) \;\;;\;\; \frac{0}{X} \longrightarrow 0 \quad (0/x) \;\;;\;\; \frac{1}{X} \times \frac{1}{Y} \longrightarrow \frac{1}{X \times Y} \quad (1/\times)$$

Figure 4.1: Symbolic differentiation CRS.

$$\mathrel{\#\!\!\!\longrightarrow} \lambda x.\underset{(\beta)}{\boxed{\left(\lambda x.\frac{1}{x}\right)(x+1)}} \times \left(\underset{(\text{identity})}{\boxed{(D([x]x, x))}} + \underset{(\text{constant})}{\boxed{(D([x]1, x))}}\right)$$

$$\mathrel{\#\!\!\!\longrightarrow} \lambda x.\frac{1}{x+1} \times \underset{(x{+}0)}{\boxed{(1+0)}} \longrightarrow \lambda x.\underset{(x1)}{\boxed{\frac{1}{x+1} \times 1}} \longrightarrow \lambda x.\frac{1}{x+1}$$

(notice the need for parentheses around ln in applications because the CRS world is 'untyped' so we cannot know whether $\ln(E)$ is a construction of $\ln^1$ *on* E or an application of $\ln^0$ *to* E).

Here is a more complex reduction where we use the 'chain rule' nontrivially, exploiting the power of higher order rewriting.

$$\underset{(\text{function})}{\boxed{d(\lambda z.(\lambda x.(\ln)(x+1))((\lambda x.\ln x)z))}}$$

$$\longrightarrow \lambda x.\underset{(\text{chain})}{\boxed{D([x](\lambda b.(\ln)(b+1))((\lambda c.\ln c)x), x)}}$$

$$\longrightarrow \lambda x.\left(\underset{(\text{function})}{\boxed{d(\lambda a.(\ln)(a+1))}}\right)\left(\underset{(\beta)}{\boxed{(\lambda e.\ln e)x}}\right) \times \underset{(\text{chain})}{\boxed{(D([x](\lambda e.\ln e)x, x))}}$$

$$\mathrel{\#\!\!\!\longrightarrow} \lambda x.\underset{(\beta)}{\boxed{(\lambda x.D([x](\ln)(x+1), x))(\ln x)}} \times \left(\left(\underset{(\text{function})}{\boxed{d(\lambda a.\ln a)}}\right) x \times \underset{(\text{identity})}{\boxed{(D([x]x, x))}}\right)$$

$$\mathrel{\#\!\!\!\longrightarrow} \lambda x.\underset{(\text{chain})}{\boxed{(D([b](\ln)(b+1), \ln x))}} \times \underset{(x1)}{\boxed{((\lambda x.D([x]\ln x, x))x \times 1)}}$$

$$\mathrel{\#\!\!\!\longrightarrow} \lambda x.\left(\left(\underset{(\ln)}{\boxed{d(\ln)}}\right)(\ln x+1) \times \underset{(\text{plus})}{\boxed{(D([x]x+1, \ln x))}}\right) \times \underset{(\beta)}{\boxed{(\lambda a.D([b]\ln b, a))x}}$$

$$\mathrel{\#\!\!\!\longrightarrow} \lambda x.\left(\underset{(\beta)}{\boxed{\left(\lambda x.\frac{1}{x}\right)(\ln x+1)}} \times \left(\underset{(\text{identity})}{\boxed{(D([x]x, \ln x))}} + \underset{(\text{constant})}{\boxed{(D([x]1, \ln x))}}\right)\right) \times \underset{(\text{chain})}{\boxed{(D([d]\ln d, x))}}$$

$$\mathrel{\#\!\!\!\longrightarrow} \lambda x.\left(\frac{1}{\ln x+1} \times \underset{(x{+}0)}{\boxed{(1+0)}}\right) \times \left(\left(\underset{(\ln)}{\boxed{d(\ln)}}\right) x \times \underset{(\text{identity})}{\boxed{(D([x]x, x))}}\right)$$

$$\mathrel{\#\!\!\!\longrightarrow} \lambda x.\underset{(x1)}{\boxed{\left(\frac{1}{\ln x+1} \times 1\right)}} \times \underset{(x1)}{\boxed{\left(\left(\lambda x.\frac{1}{x}\right) x \times 1\right)}} \mathrel{\#\!\!\!\longrightarrow} \lambda x.\frac{1}{\ln x+1} \times \underset{(\beta)}{\boxed{\left(\lambda a.\frac{1}{a}\right) x}}$$

$$\longrightarrow \lambda x.\underset{(1/\times)}{\boxed{\frac{1}{\ln x+1} \times \frac{1}{x}}} \longrightarrow \lambda x.\frac{\underset{(1x)}{\boxed{1 \times 1}}}{(\ln x+1) \times x} \longrightarrow \lambda x.\frac{1}{(\ln x+1) \times x}$$

4.2 Explicit Substitutes

In this section we show how the explicit substitution idea developed for the λ-calculus in section 1.1 can be generalised to CRS systems, and in particular we show that there is a translation from an explicit substitution CRS to the original CRS.

This brings us part of the way towards being able to use the CRS reduction count as a complexity measure in that we eliminate 'deep substitution'. The technique used is the same as for the λ-calculus: to perform substitution in a stepwise manner, or, put differently, to change metaredexes to use 'explicit substitutes', such that only *local term knowledge* is used in the rewriting.

First we demonstrate the idea as it was developed for a particular CRS, namely the $\lambda\beta$-calculus (essentially recalling section 1.1 in the CRS notation). Then we identify a subclass of CRSs that has 'explicit substitution' and show that every confluent CRS can be transformed into a confluent CRS in this class.

4.2.1 Example (λxgc-reduction as CRS). λxgc is the CRS over terms

$$t \;::=\; x \;\mid\; \lambda x.t \;\mid\; t_1 t_2 \;\mid\; \Sigma([x]t_1, t_2)$$

with λxgc-CRS rules

$$(\lambda x.Z(x))Y \to \Sigma([x]Z(x), Y) \tag{b}$$
$$\Sigma([x]x, Y) \to Y \tag{xv}$$
$$\Sigma([x]Z, Y) \to Z \tag{xgc}$$
$$\Sigma([x]\lambda y.Z(x,y), Y) \to \lambda y.\Sigma([x]Z(x,y), Y) \tag{xab}$$
$$\Sigma([x]\,(Z_1(x))(Z_2(x)), Y) \to (\Sigma([x]Z_1(x), Y))(\Sigma([x]Z_2(x), Y)) \tag{xap}$$

(as ususal we use x to refer to just the rules with names starting with x). It is instructive to compare this system to the definition of λxgc in Definition 1.1.4: they are identical except for syntax conventions (for instance the λxgc-term $x\langle x := \lambda y.y\rangle$ corresponds to $\Sigma([x]x, \lambda y.y)$).

Furthermore, the requirement that all metaterms are closed prohibits free variables in CRS rules, which means that all free variables must be 'hidden' inside metavariables (to which they are not given as parameters). This means that we cannot distinguish between substituting a free variable (λxgc rule (xvgc)) and the more general garbage collection (λxgc rule (gc)): both are special cases of the rule (xgc) above.

This is not an orthogonal CRS because it has the following overlap patterns:

$$\underbrace{\boxed{\Sigma\left([x]\underbrace{\boxed{(\lambda a.Z(a,x))(Z_2(x))}}_{(b)},Y\right)}}_{(xap)} \qquad \underbrace{\boxed{\underbrace{\boxed{\Sigma([x]\lambda a.X(a),Y)}}_{(xab)}}}_{(xgc)} \qquad \underbrace{\boxed{\underbrace{\boxed{\Sigma([x]Z_1Z_2,Y)}}_{(xap)}}}_{(xgc)}$$

Yet it is confluent as proven in Bloo and Rose (1995) and as a consequence of the following.

The key observation in the development for the λ-calculus from section 1.1is that no knowledge of the 'depth' of terms is needed in order to reduce the rules. In the example rules above this is manifest in the fact that all metaapplications on both LHS and RHS have the *same variables on both sides* which means that substitution can be realised purely locally. This is formalised as follows.

4.2.2 Definition (explicit substitution CRS). A CRS R is an *explicit substitution CRS* or simply *ESCRS*, if all metaapplications in the RHS of a rule occur in the form $Z(\vec{x})$ such that $Z(\vec{x})$ also occurs in the LHS of that same rule.[3] The individual metaapplications in RHSs obeying this will be called *explicit metaapplications.*

Since all metavariables of a CRS rule must occur in the LHS in the form $Z(\vec{x})$ this definition means that each metaapplication of Z in that rule is metaapplied to the same list of variables as in the LHS. Recall that the intention is to avoid having to perform substitution. Here are some small sample CRSs that are and are not explicit.

4.2.3 Examples (explicit and nonexplicit CRSs). Consider CRSs over the alphabet A^1, B^2, C^0.

1. $A([x,y]B(X(x),Y)) \to X(Y)$ *is not* explicit because we have to substitute Y for any occurrence of x in whatever matched $X(x)$.
2. $A([x,y]B(x,y)) \to A([x,y]B(C,x))$ *is* explicit because it contains no metaapplications!
3. $A([x,y]A(X(x,y))) \to A([x]X(x,x))$ *is not* explicit because we need to change one of the variables *inside* what matched $X(x,y)$.
4. $A([x,y]A(X(x,y))) \to A([x,y]A(X(y,x)))$ *is* explicit because we can just rearrange the variables so we do not have to change anything inside what

[3]This implies that the variables in $\vec{x}$ are distinct.

macthed $X(x,y)$ (this is perhaps more obvious if the rule is reformulated as $A[x,y]A(X(x,y)) \to A[y,x]A(X(x,y))$ but names of bound variables do not matter in CRS systems).

5. $A([x,y]A(X(x,y))) \rightarrowtail A([x,y]B(X(x,y),X(y,x)))$ *is not* explicit because in the second we need to change one of the variables *inside* what matched $X(x,y)$.

6. $A([x,y]B(X(x,y),Y(x,y))) \to A([x,y]B(X(x,y),Y(y,x)))$ *is not* explicit for the same reason.

7. $A([x,y]B(X(x,y),X(y,x))) \to A([x,y]B(X(x,y)))$ *is* explicit.

8. $C([x]X(x),[y]X(y)) \to A([x]X(x))$ *is* explicit because we can just *choose* to use the bound variable that we know to be in the X-metaapplication we matched.

9. $A([x]X(x)) \to B([x,y]X(x),[y]X(y))$ *is not* explicit because in *one* of the X-constructions in the contractum we will have to substitute in a new variable.

4.2.4 Theorem. *An explicit CRS is local.*

Proof. Each rule in such a CRS can clearly be described as a *rearrangement of local symbols* and all subterms can be moved as units. ☐

However, in order to achieve a notion of reduction with a *realistic complexity* then we wish the work involved in each reduction to be bound by some constant that does not depend on the term being reduced. The main problem of the complexity of CRS reduction is the same as for the λ-calculus, namely *substitution* as discussed in section 1.1 – this is what we have already handled. However, the *time of pattern matching* also plays a rôle:

- If a pattern is *non-linear* then pattern matching requires time proportional to the size of the subterms that need to be identical in the redex. This observation is well-known.

- If a pattern contains *metaapplications applied to variables lists that do not contain all the bound variables* then it is necessary to check in the redex whether only the permitted bound variables occur in the redex – again this takes time proportional to the size of the redex. This calls for a new concept.

4.2.5 Definition (saturated). A metaterm is *saturated* if for every metaapplication $Z(\vec{t})$ the set of free variables $\mathrm{fv}(Z(\vec{t}))$ includes all variables bound at the occurrence of the metaapplication.

4.2.6 Examples. (β) is saturated, (η) of Remark A.3.9 is not, neither is the λxgc-CRS shown in Example 4.2.1 because of (xgc).

4.2.7 Remark (free variable matching). In order to be able to handle λxgc in its original form we introduce the following notion: a *free variable pattern* is a special metaapplication form Z^{ν} that only matches variables not otherwise bound by the match. This allows us to write the (xvgc) rule as

$$\Sigma([x]Z^{\nu}, Y) \to Z$$

exploiting that Z^{ν} cannot contain x (because then we should have written $Z^{\nu}(x)$) and at the same time it must be a variable, hence it can only match variables $x \neq y$. We have implemented this in the program in Rose (1996b, chapter 6). Using this rule instead of (xgc) of course removes the two last overlaps.

4.2.8 Discussion (fair complexity measure?). We have achieved what we set out to do: the reduction count is a fair complexity measure *provided we only count matching and contraction complexity*: we have not considered the cost of matching. We will return to the redex search complexity in the next section.

Another interesting observation which we can carry over from the study of explicit substitution for the λ-calculus is that substitution does not depend on the context in which it is created. This insight can be used to *create an ESCRS automatically from a CRS* in a manner that ensures that confluence is preserved by simply 'unfolding' the definition of substitutes (Definition 4.1.8) into the new rules.

4.2.9 Definition (CRS explicification). Given a CRS R with alphabet F_i^n. The ESCRS Rx is obtained by the steps listed in Figure 4.2. If R defined the relation $\to$ then we will denote the relation defined by Rx as $\underset{\mathrm{ie}}{\to}$; the subrelation consisting of only the introduction rules (with name (r-x)) is denoted $\underset{\mathrm{i}}{\to}$, and the subrelation containing only the substitution distribution and elimination rules (with names of form (x-_)) is denoted $\underset{\mathrm{e}}{\to}$; we write $\xrightarrow[\mathrm{xgc}]{}$ for the union of all (xgc-n-i) rules. Note that there are no rules for interaction between two substitutions.

Restricted terms. The restricted terms of the CRS Rx are defined by extending the syntax for R with the clause $\Sigma^{n+1}(\,[\vec{x}]t, \vec{t}\,)$ where the symbols $\Sigma^2, \ldots$ are added to the alphabet as needed in the substitution introduction rules below. Terms that do not contain subterms of the form $\Sigma^{n+1}([\vec{x}]s, \vec{t})$ will be called *pure*.

Substitution introduction. For each rule (r) of R construct a new rule (r-x) of Rx by replacing in the RHS all non-explicit $Z^n(\vec{t})$ by

$$\Sigma^{n+1}([\vec{x}]Z^n(\vec{x}_{(n)}), \vec{t}_{(n)}\,)$$

Hence the arities of non-explicit metaapplications decide which new function symbols we need.

Stepwise substitution distribution. For each Σ^{n+1} symbol add a rule

$$\Sigma^{n+1}([\vec{x}][y]Z(\vec{x},y), \vec{X}) \to [y]\Sigma([\vec{x}]Z(\vec{x},y), \vec{X}) \tag{xma-n}$$

and for each possible Σ^{n+1}, F^m pair add a rule

$$\begin{aligned}&\Sigma^{n+1}([\vec{x}]F^m(Z_1(\vec{x}), \ldots, Z_m(\vec{x})), \vec{X}) \\ &\qquad \to F^m(\Sigma^{n+1}([\vec{x}]Z_1(\vec{x}), \vec{X}), \ldots, \Sigma^{n+1}([\vec{x}]Z_m(\vec{x}), \vec{X}))\end{aligned} \tag{x-F^m-n}$$

Substitution elimination. For each Σ^{n+1} add for each $i \in \{1, \ldots, n\}$ two rules

$$\Sigma^{n+1}([\vec{x}]x_i, \vec{X}) \to X_i \tag{xv-n-i}$$

$$\Sigma^{n+1}([\vec{x}]Z(\vec{x}\,'), \vec{X}) \to \begin{cases} Z & \text{if } n = 1 \\ \Sigma^n([\vec{x}\,']Z(\vec{x}\,'), \vec{X}\,') & \text{if } n > 1 \end{cases} \tag{xgc-n-i}$$

$$\text{where} \quad \begin{aligned}\vec{x}\,' &= (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \\ \vec{X}\,' &= (X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n)\end{aligned}$$

(thus the LHS of (xgc-n-i) includes the abstraction for x_i in the variable list and the corresponding element X_i of the substitution body but not in the parameter list $Z(\vec{x}\,')$; the RHS excludes x_i in the metaabstraction and X_i in the right hand side).

Figure 4.2: Explicification of CRS R into ESCRS Rx.

4.2.10 Example. The system $\lambda\beta x$ generated this way for the $\lambda\beta$-calculus with the rule

$$(\lambda x.Z(x))Y \to Z(Y) \tag{β}$$

is the following:[4]

$$\begin{aligned}
(\lambda x.Z(x))Y &\to \Sigma([x]Z(x), Y) && (\beta\text{-x})\\
\Sigma([x](Z_1(x))(Z_2(x)), Y) &\to (\Sigma([x]Z_1(x), Y))(\Sigma([x]Z_2(x), Y)) && (\text{x-@})\\
\Sigma([x]\lambda(Z(x)), Y) &\to \lambda(\Sigma([x]Z(x), Y)) && (\text{x-}\lambda)\\
\Sigma([x]x, Y) &\to Y && (\text{xv})\\
\Sigma([x]Z, Y) &\to Z && (\text{xgc})\\
\Sigma([x, y]Z(x, y), Y) &\to [y]\Sigma([x]Z(x, y), Y) && (\text{xma})
\end{aligned}$$

It is the same as the λxgc CRS shown above except that the abstraction distribution rule (xab) has been split into two steps: (x-λ) and (xma). As a consequence, a restricted term need not reduce to a restricted term (but another reduction step can fix this). To prevent this, one can define the explicification more cautiously for CRSs with restricted terms (this means that some $\underset{e}{\to}$ steps are forced to be followed by other $\underset{e}{\to}$ steps). We will not do this in this tutorial since it is rather straightforward.

4.2.11 Proposition. For any CRS R, Rx is explicit.

4.2.12 Remark. The procedure is *idempotent*: when applied to an ESCRS nothing is added since no non-explicit metaabstractions exist. Also notice that the elimination and distribution rules only depend on the alphabet.

The remainder of this section is devoted to showing that the derived ESCRS is a conservative extension of the original CRS.

4.2.13 Proposition. For a confluent CRS $\to$, $\underset{e}{\to}$ is confluent and strongly normalising.

Proof sketch. Completeness of $\underset{e}{\to}$ is shown by first establishing $\underset{e}{\to}$SN by defining a map dominating the longest reduction length similarly as for λxgc in Bloo and Rose (1995); $\underset{e}{\to}$LC follows from a simple investigation of the critical pairs between the (xgc-n) rules and the other rules (this amounts to understanding that 'garbage collection' can be postponed without risk). □

[4] Generated automatically from (β) by the program of Rose (1996b, chapter 6).

It is not difficult to extend this result to reduction on metaterms.

Next we relate single reductions and then build the components of the proof of multiple reduction equivalence (in this paper $\downarrow_e(t)$ denotes the unique $\underset{e}{\rightarrow}$-nf of t, and $\underset{e}{\twoheadrightarrow\!\!\!\!\!\!\twoheadrightarrow}$ the restriction of $\underset{e}{\twoheadrightarrow}$ to reductions to normal forms).

4.2.14 Lemma (representation). For terms t, t_1,

$$\downarrow_e(\Sigma([\vec{x}]t, \vec{t})) \equiv \downarrow_e(t)[\vec{x} := \downarrow_e(\vec{t})]$$

(where we mean parallel substitution).

Proof. Note that t and t_1 do not contain metavariables. This lemma is the 'substitution lemma' of the λ-calculus in disguise; we prove by induction on the number of symbols in the sequence $t, t_1, \ldots, t_n$ that $\downarrow_e(\Sigma([x_1, \ldots, x_n]t, t_1, \ldots, t_n)) \equiv \downarrow_e(t)[x_1 := \downarrow_e(t_1)] \cdots [x_n := \downarrow_e(t_n)]$. We distinguish some cases according to the structure of t.

Case $t \equiv x_i$. Then $\Sigma([\vec{x}]t, \vec{t}) \underset{e}{\twoheadrightarrow} \Sigma([x_{i+1}, \ldots, x_n]t_i, t_{i+1}, \ldots, t_n)$, hence

$$\begin{aligned}
\downarrow_e(\Sigma([\vec{x}]t, \vec{t})) &\equiv \downarrow_e(\Sigma([x_{i+1}, \ldots, x_n]t_i, t_{i+1}, \ldots, t_n)) \\
&\downarrow_e(\Sigma([x]t, \vec{t})) \\
&\equiv \downarrow_e(\Sigma([x_{i+1}, \ldots, x_n]t_i, t_{i+1}, \ldots, t_n)) \\
&\overset{\text{IH}}{\equiv} \downarrow_e(t_i)[x_{i+1} := \downarrow_e(t_{i+1})] \cdots [x_n := \downarrow_e(t_n)] \\
&\equiv \downarrow_e(t)[x_1 := \downarrow_e(t_1)] \cdots [x_n := \downarrow_e(t_n)]
\end{aligned}$$

Case $t = F^m(\vec{s})$. Then $\Sigma([\vec{x}]t, \vec{t}) \underset{e}{\twoheadrightarrow} F^m(\Sigma([\vec{x}]s_1, \vec{t}), \ldots, \Sigma([\vec{x}]s_m, \vec{t}))$, hence

$$\begin{aligned}
\downarrow_e(\Sigma([\vec{x}]t, \vec{t})) &\equiv F^m(\downarrow_e(\Sigma([\vec{x}]s_1, \vec{t})), \ldots, \downarrow_e(\Sigma([\vec{x}]s_m, \vec{t}))) \\
&\overset{\text{IH}}{\equiv} F^m(\downarrow_e(s_1)\overrightarrow{[x_i := \downarrow_e(t_i)]}, \ldots, \downarrow_e(s_m)\overrightarrow{[x_i := \downarrow_e(t_i)]}) \\
&\equiv (F^m(\downarrow_e(s_1), \ldots, \downarrow_e(s_m)))\overrightarrow{[x_i := \downarrow_e(t_i)]} \\
&\equiv \downarrow_e(F^m(\vec{s}))\overrightarrow{[x_i := \downarrow_e(t_i)]}
\end{aligned}$$

Case $t \equiv \Sigma([x]t', t'')$. Now use IH on $\Sigma([x, x_1, \ldots, x_n]t', t'', t_1, \ldots, t_n)$. □

4.2.15 Lemmas (projection & injection).

1. For terms s, t, $\begin{tikzcd} s \arrow[r, "i"'] \arrow[d, two heads, "e"] & t \arrow[d, two heads, "e"] \\ \downarrow_e(s) \arrow[r, dotted, two heads] & \downarrow_e(t) \end{tikzcd}$. 2. For pure terms s, t, $\begin{tikzcd} s \arrow[rr] \arrow[dr, dotted, "i"'] & & t \\ & s' \arrow[ur, dotted, two heads, "e"'] & \end{tikzcd}$.

Proof. Lemma 4.2.15.1 by induction over the structure of Rx-metaterms, using Lemma 4.2.14; Lemma 4.2.15.2 is a simple consequence of the fact that the R-rules and Rx-introduction rules have the same redexes and the $\underset{e}{\rightarrow}$-rules construct no new redexes. The details are as follows:

1. Induction over the structure of s. If the redex contracted in $s\underset{i}{\rightarrow}$ is at the root of s, then use Lemma 4.2.14. If $s \equiv F^m(\vec{s})$ or $s \equiv [x]s_i$ and the redex is in s_i then use the induction hypothesis on s_i. Otherwise, $s \equiv \Sigma([\vec{x}]s', \vec{s})$ and the redex is in either s or one of the s_i. Then by Lemma 4.2.14, $\downarrow_e(s) \equiv \downarrow_e(s')[\vec{x} := \downarrow_e(\vec{s})]$ and by the induction hypothesis $\downarrow_e(s_i) \twoheadrightarrow \downarrow_e(t_i)$ if $t \equiv \Sigma([x]t', \vec{t})$.

2. If $s \rightarrow t$ contracts the redex at the root of s then by the corresponding introduction-rule we can contract the redex at the root of s in $s \underset{i}{\rightarrow} s'$. Now by Lemma 4.2.14 and the definition of substitution in Definition 4.1.8 it is clear that $\downarrow_e(s') \equiv t$, hence $s' \underset{e}{\twoheadrightarrow} t$. ☐

4.2.16 Theorem. *For* R*-terms* s, t, $s \underset{ie}{\twoheadrightarrow} t$ *iff* $s \twoheadrightarrow t$.

Proof. First observe that the R-terms s, t are in e-normal form when considered as Rx-terms. Then use Lemma 4.2.15:

Case $\Leftarrow$: Assume $s \twoheadrightarrow t$; the case then follows by induction on the length of the $\twoheadrightarrow$-reduction, using Lemma 4.2.15.2 in each step.

Case $\Rightarrow$: Assume $s \underset{ie}{\twoheadrightarrow} t$ and $s, t \in R$. We will do induction on the length of the $\underset{ie}{\twoheadrightarrow}$-reduction and prove

$$\begin{tikzcd} s \arrow[r, "ie"'] \arrow[d, equal] & s_1 \arrow[r, "ie"'] \arrow[d, two heads, "e"'] & \cdots \arrow[r, "ie"'] & s_{n-1} \arrow[r, "ie"'] \arrow[d, two heads, "e"'] & t \arrow[d, equal] \\ s \arrow[r, dotted, two heads] & \downarrow_e(s_1) \arrow[r, dotted, two heads] & \cdots \arrow[r, dotted, two heads] & \downarrow_e(s_{n-1}) \arrow[r, dotted, two heads] & t \end{tikzcd}$$

Each $\underset{ie}{\rightarrow}$-step is either $\underset{i}{\rightarrow}$ or $\underset{e}{\rightarrow}$ for which we need Lemma 4.2.15.1 and confluence of $\underset{e}{\rightarrow}$, respectively (in the latter case $\twoheadrightarrow$ is void). ☐

An easy consequence of all this is the main result of this section.

4.2.17 Corollary. *If* R *is confluent then* Rx *is confluent.*

Proof. If $t \twoheadrightarrow_{ie} s_1$ and $t \twoheadrightarrow_{ie} s_2$ then by Lemma 4.2.15.1 we have both $\downarrow_e(t) \twoheadrightarrow \downarrow_e(s_1)$ and $\downarrow_e(t) \twoheadrightarrow \downarrow_e(s_2)$, now use confluence of $\twoheadrightarrow$ and Theorem 4.2.16. □

Finally, we mention a new result of Roel Bloo: it turns out that the construction above *preserves strong normalisation* for an interesting subclass of CRSs. We quote from Bloo and Rose (1996):

4.2.18 Definition. A CRS is called *structure preserving* if any argument of a metaapplication in the RHS of a rule is a subterm of the LHS of that rule.

4.2.19 Theorem. R*x PSN of* R.

Proof. See Bloo and Rose (1996). □

4.3 Explicit Addresses

In this section we show how the explicit sharing idea developed for the λx-calculus in section 3.2 can be generalised to explicit CRSs with the restrictions discussed in the previous section.

In the previous sections, we have obtained a description of CRS systems that have efficient pattern matching through restrictions on the form of patterns, and efficient term traversal through the use of explicit substitution. What is missing is to have a proper operational description of *space usage* just as we obtained for λ-calculus.

In this section we generalise the address notion of section 3.1 to CRSs. The generalisation will be separated into two cases. The first case is *acyclic sharing*, which is a smooth and systematic generalisation of Wadsworth's λ-graph sharing that is stable with respect to a large class of reduction strategies. The second case is what we will call *semi-cyclic sharing* which is a synthesis of explicit substitution and Turner's (1979) graph combinators.

4.3.1 Definition (simple sharing CRS). The set of *simply addressed CRS-preterms* over some alphabet is the set of preterms (as in Definition 4.1.1.2) extended with *addressing* forms (as in Definition 3.1.1.1), *i.e.*,

$$t ::= x \mid [x]t \mid F^n(t_1, \ldots, t_n) \mid Z^n(t_1, \ldots, t_n) \mid t^a$$

where abc range over an infinite set of addresses and we permit *free addresses* in rule RHSs: each occurrence of a free address denotess "a *fresh address* globally guaranteed unique"; the rest is as in Definition 4.1.1. Matching and rewriting also proceeds quite as in Definition 4.1.12 with addresses seen as new function symbols, except that the relatiosn we define will allways implicitly be the sharing closure of what is asked for. We call such a system a 'CRSa' where the 'a' stands for 'addressed'.

The fact that the 'extension' can also be seen as a restriction on the terms makes the definition particularly simple – in fact all the definitions of section 3.1 up to Proposition 3.1.6 carry over immediately to the above class, and we will use them freely. The only thing that is not internalisable is allocation of fresh addresses for which we will assume and refer to an *address oracle*. However, remark that such a device can, in fact, be efficiently implemented on a computer. Further, the relation enjoys the pleasant properties of a sharing as discussed in section 3.1.

4.3.2 Example (acyclic λ-graph reduction). The CRSa

$$(\lambda x.Y(x))X \to Y((X)^{\text{fresh}}) \tag{βa}$$

$$(\lambda x.Y(x))^Z \to (\lambda x.Y(x)) \tag{copy}$$

defines λ-calculus with sharing. To illustrate its use, here is the reduction of the λ-term $(\lambda k.Skk)(IIK)$ with leftmost and sharing reduction.

$$\underset{(\beta a)}{\boxed{(\lambda k.(\lambda abc.ac(bc))kk)(((\lambda a.a)(\lambda a.a))(\lambda ab.a))}}$$

$$\downarrow$$

$$\underset{(\beta a)}{\boxed{((\lambda abc.ac(bc))((((\lambda a.a)(\lambda a.a))(\lambda ab.a))^1\,))}}((((\lambda a.a)(\lambda a.a))(\lambda ab.a))^1\,)$$

$$\downarrow$$

$$\underset{(\beta a)}{\boxed{(\lambda ab.(((((\lambda a.a)(\lambda a.a))(\lambda ab.a))^1\,)^2\,)b(ab))((((\lambda a.a)(\lambda a.a))(\lambda ab.a)}}\,)^1\,)$$

$$\downarrow$$

$$\lambda a.(((\underset{(\beta a)}{\boxed{((\lambda b.b)(\lambda b.b))}}(\lambda bc.b))^1\,)^2\,)a((((((\lambda a.a)(\lambda a.a))(\lambda ab.a))^1\,)^3\,)a)$$

$$\downarrow$$

$$\lambda a.((((\underset{(\text{copy})}{\boxed{(\lambda b.b)^4}})(\lambda bc.b))^1\,)^2\,)a((((((\lambda b.b)^4\,)(\lambda bc.b))^1\,)^3\,)a)$$

$$\downarrow$$

$$\lambda a.(((\underset{(\beta a)}{\boxed{(\lambda b.b)(\lambda bc.b)}})^1\,)^2\,)a(((((\lambda b.b)(\lambda bc.b))^1\,)^3\,)a)$$

$$\downarrow$$

$$
\begin{gathered}
\lambda a.(((\underset{(\text{copy})}{\boxed{(\lambda bc.b)^{6}}})^{1})^{2})a(((((\lambda bc.b)^{6})^{1})^{3})a) \\
\downarrow \\
\lambda a.((\underset{(\text{copy})}{\boxed{(\lambda bc.b)^{1}}})^{2})a((((\lambda bc.b)^{1})^{3})a) \\
\downarrow \\
\lambda a.(\underset{(\text{copy})}{\boxed{(\lambda bc.b)^{2}}})a((((\lambda bc.b)^{1})^{3})a) \\
\downarrow \\
\lambda a.\underset{(\beta a)}{\boxed{(\lambda bc.b)a}}((((\lambda bc.b)^{1})^{3})a) \\
\downarrow \\
\lambda a.\underset{(\beta a)}{\boxed{(\lambda b.(a)^{10})((((\lambda bc.b)^{1})^{3})a)}} \\
\downarrow \\
\lambda a.(a)^{10}
\end{gathered}
$$

As for the λ-calculus systems with sharing do not work properly without a *copying* rule, and care has to be exercised when reducing. Fortunately most such problems disappear when combining with explicit substitution.

4.3.3 Definition (explicit substitution and sharing CRS). Given a CRSa, R. The result of explicifying this, Rx, will be called an ESCRSa.

For such a system everything works out nicely, just as for the λ-calculus in section 3.2: since all rewrites are local no undesired 'side effects' happen in the term. We will demonstrate this, but first we will strenthen the notion of sharing.

4.3.4 Discussion (explicit recursion). The classical observation that

" explicit recursion = cyclic references "

has been used since the LABEL primitive of LISP which was implemented by constructing a cyclic data structure. However, the construction gives theoretical headaches because of the following observation, freely after (Kennaway, Klop, Sleep and de Vries 1995)

" true cycles correspond to infinitary reductions "

Essentially this observation means that if true cycles are included in rewriting then locality breaks down in an essential way. For several examples of this as well as an elegant but non-compositional solution see Ariola and Klop (1994) (more on cyclic λ-graphs below).

We will instead use a 'representation trick' also used by Felleisen and Friedman (1989) for implementing selective updating: we will introduce an explicit *back-pointer* node which is allowed to have an address that occurs on the path to it from the root. This means we have to weaken the wfa property to allow it, of course – it looks like this.

4.3.5 Definition (cyclic sharing CRS). The set of *cyclic addressed CRS-preterms* over some alphabet is the set of simple sharing preterms (as in Definition 4.3.1) extended with *cyclic addressing* forms, *i.e.*,

$$t ::= x \mid [x]t \mid F^n(t_1, \ldots, t_n) \mid Z^n(t_1, \ldots, t_n) \mid t^a \mid \bullet^a$$

Rewriting happens as for CRSa except we add another metanotation that is allowed in rule RHSs, namely $t_1 [\,] t_2^a$ which means $t_1[t_2^a]$ in the notation of Definition 3.1.2.2 which *must be used whenever an address is removed that may occur in a subterm*.

We will call such a system a 'CRSar' where the 'r' stands for 'recursive'.

The importance of the update operation is highlighted by the following.

4.3.6 Definitions (cyclic sharing terms). Assume a set of adressed T-terms as above.

1. We write 't wfar' if t is an addressed term where all subterms with the same address are identical or a nested $\bullet$-node: for all subterms t_1^a, t_2^a of t either

    1. $t_1 = t_2$ (thus both either $\bullet$ or not), or
    2. $t_1 = \bullet$ and $t = C[(C'[\bullet^a])^a]$ such that $(C'[\bullet^a])^a = t_2$ (or similarly with t_1, t_2 swapped).

2. The *updating* of t wfar with s wfar at address $a \notin \mathrm{addr}(s)$ is written $t[s^a]$ and is obtained by replacing all *outermost* subterms $u^a \trianglerighteq t$ (if any) by s^a. By convention $t[\epsilon] \equiv t$ (to ensure $t[t@a] = t$ for all a).

3. The set of *cyclic sharing* T*-terms*, notation T wfar, is the subset of the addressed T-terms with wfar.

When a term is wfar we will say for each address in it that "the" subterm $t^a \neq \bullet^a$ is *shared* with address a; if furthermore $\bullet^a \triangleright t$ then it is *cyclic*.

4.3.7 Example (cyclic λ-graph reduction). The CRSa

$$(\lambda x.Y(x))X \to Y((X)^{\text{fresh}}) \tag{βa}$$

$$(\lambda x.Y(x))^{Z} \to (\lambda x.Y(x)) \tag{copy}$$

defines λ-calculus with sharing and explicit recursion. It is easy to verify that wfar is preserved

So when is it we wish to *use* a cyclic definition? The just shown example reveals the kind of rule that is susceptible to recursion: the μ-rule $\mu x.Z(x) \to Z(\mu x.Z(x))$ contains *the LHS as an argument to a construction of the RHS* – it is a kind of *self-application*.

We will give several examples in section 4.4.

4.3.8 Comparison (cyclic λ-graph reduction). The λ-graph notion of Ariola and Klop (1994) is the main related work on cyclic sharing with reduction which is similar to explicit substitution and independent of reduction strategies; that work and this tutorial subsume the earlier *Graph rewrite systems* (GRS) of Ariola and Arvind (1992) and *cyclic substitutions* of Rose (1992)); all these origin with Wadsworth (1971) where cycles are discussed but ruled out.

The main difference between our work and Ariola and Klop's (1994) λ-graphs is that *cycles remain observable*: it is possible to write rules that manipulate •s, for example, and the CRSar have to decide when to unfold what Ariola and Klop (1994) call 'vertical sharing' manually. The gain is that our system is confluent: the wfar property forces a certain synchronisation between subterms that is sufficient to ensure that the *recursion variable besting depth* by Ariola and Klop (1995b) is always finite.

4.4 CRS and Functional Programs

In this section we discuss the connection between CRS and functional programming.

Our starting point is the idea that

“ functional programming = term rewriting ”

based on the simple established connection from *first order functional programming* to *recursive equations* since these can be *directed* (according to the Church-Rosser Theorem A.3.11 theorem, in fact) and thus interpretable as term rewrite systems (TRS). It is known that when this idea is extended with a notion of sharing then it is sufficiently refined to serve as a basis even for implementations of functional languages as has indeed been done by Turner (1979) and studied by Barendregt et al. (1987). For first order programs this correspondence is so close that one can *relate the computation steps* – put simply,

“ 1 functional program evaluation step = 1 term rewrite step ”

However, the model fails to address one major complication of functional program evaluation, namely the behaviour of *higher order functions*: even though it is possible to transform higher order functions into rewrite rules, it is in general not clear whether this preserves operational properties. For example, it is not obvious how λ-calculus reduction can be expressed in this way.

This is where the considerations of the previous sections contribute to investigate the generalisation

“ higher order functional programming = higher order rewriting ”

We would like to demonstrates how CRS can serve admirably as the generalisation of rewriting as a model of the operational behaviour of pure functional programs including higher order functions – the converse we have already addressed: how CRS can be resolved into elementary reductions. The goal is to establish a complexity-faithful link as for the first-order case. This is difficult which is surprising considering that CRS have many of the properties of pure functional programming languages, notably *pattern matching*, *referential transparency* (which we call confluence), inherited from TRS, and *abstraction over bound variables* (inherited from the λ-calculus).

In a nutshell: what is ‘wrong’ with, say the differential rewrite system of Figure 4.1, if we want to implement it? Three things pop up as ‘expensive’, namely

non-linear patterns: when a pattern repeats a metavariable then matching involves testing for equality the appropriate subterms, this clearly takes time dependent of the size of the terms in question;

non-saturated variable lists: when a pattern metaapplication does not include the full list of available variables then we have to search the actual term to ensure that the ‘forbidden’ variables do not occur, *e.g.*, when applying rule (constant) above; and

deep substitution: the act of substitution usually takes time proportional to the size of the substitute (just as we have argued for the λ-calculus in chapter 1.

The first aspect was already defined in section 4.1, and it is already present in functional programming where patterns must also be linear. The second is also present in functional programming in that pattern variables (corresponding to the CRS metavariables) have zero arity, however, here the CRS formalism can actually do a bit more while keeping pattern matching in constant time (because no unification or higher-order matching is allowed). The third aspect is avoided by insisting on explicit systems.

In the remainder of this section we discuss a typefree variant of Plotkin's (1977) PCF programming language, extended with a pairing operator : and the associated HD and TL selectors, and show how sharing and explicification works.

The PCF+pairs programs and evaluation reduction system as a CRS are given in Figure 4.3. The following small program computes the parity of a number, or rather: the pair of booleans $(\mathit{is\text{-}even} : \mathit{is\text{-}odd})$.

$$\mu f.\lambda n.\mathrm{IF}(n=0,T,\mathrm{TL}(f(n-1))) : \mathrm{IF}(n=0,F,\mathrm{HD}(f(n-1)))$$

Here is the start and finish of a reduction sequence computing[5] the even and odd parities of $3=((0+1)+1)+1$ reducing as many redexes as possible in each reduction step (this is of no consequence as the system is orthogonal and hence confluent).

$$\underbrace{\boxed{(\mu f.\lambda n.(\mathrm{IF}(n=0,T,\mathrm{TL}(f(n-1)))) : \mathrm{IF}(n=0,F,\mathrm{HD}(f(n-1))))}}_{(\mu)}(((0+1)+1)+1)$$

$$\downarrow$$

$$\underbrace{\boxed{\begin{array}{l}(\lambda a.(\mathrm{IF}(a=0,T,\mathrm{TL}((\mu a.\lambda b.(\mathrm{IF}(b=0,T,\mathrm{TL}(a(b-1))))\\ : \mathrm{IF}(b=0,F,\mathrm{HD}(a(b-1)))))(a-1)))) :\\ \mathrm{IF}(a=0,F,\mathrm{HD}((\mu a.\lambda b.(\mathrm{IF}(b=0,T,\mathrm{TL}(a(b-1)))) :\\ \mathrm{IF}(b=0,F,\mathrm{HD}(a(b-1)))))(a-1))))(((0+1)+1)+1)\end{array}}}_{(\beta)}$$

$$\downarrow$$

$$(\mathrm{IF}(\underbrace{\boxed{(((0+1)+1)+1)=0}}_{(=0\text{-}F)},T,\mathrm{TL}($$

$$\underbrace{\boxed{(\mu a.\lambda b.(\mathrm{IF}(b=0,T,\mathrm{TL}(a(b-1)))) : \mathrm{IF}(b=0,F,\mathrm{HD}(a(b-1))))}}_{(\mu)}$$

$$\underbrace{\boxed{((((0+1)+1)+1)-1)}}_{(-1)}))) : \mathrm{IF}(\underbrace{\boxed{(((0+1)+1)+1)=0}}_{(=0\text{-}F)},F,\mathrm{HD}($$

$$\underbrace{\boxed{(\mu a.\lambda b.(\mathrm{IF}(b=0,T,\mathrm{TL}(a(b-1)))) : \mathrm{IF}(b=0,F,\mathrm{HD}(a(b-1))))}}_{(\mu)}\underbrace{\boxed{((((0+1)+1)+1)-1)}}_{(-1)}))$$

[5] Computed by the program of Rose (1996b, chapter 6) acting on Figure 4.3.

Syntax. of program *expressions*:

$$
\begin{aligned}
e \;::=\;& 0 \mid e+1 \mid e-1 \mid (e:e') \mid \textsc{hd}(e) \mid \textsc{tl}(e)\\
\mid\;& \mathsf{T} \mid \mathsf{F} \mid e=0 \mid \textsc{if}(e,e',e'')\\
\mid\;& x \mid \lambda x.e \mid e\,e' \mid \mu x.e
\end{aligned}
$$

Reduction.

$$(\lambda x.Z(x))X \to Z(X) \tag{β}$$

$$\mu x.Z(x) \to Z(\mu x.Z(x)) \tag{μ}$$

$$\textsc{if}(\mathsf{T},X,Y) \to X \tag{if-T}$$

$$\textsc{if}(\mathsf{F},X,Y) \to Y \tag{if-F}$$

$$0=0 \to \mathsf{T} \tag{=0-T}$$

$$(X+1)=0 \to \mathsf{F} \tag{=0-F}$$

$$((X+1)-1) \to X \tag{-1}$$

$$\textsc{hd}(X:Y) \to X \tag{hd}$$

$$\textsc{tl}(X:Y) \to Y \tag{tl}$$

Figure 4.3: Plotkin's PCF+pairs as CRS.

$$
\begin{array}{c}
\Downarrow \\
\vdots \\
\Downarrow \\
(\textsc{if}(\underset{(=0\text{-T})}{\boxed{0=0}}, \mathsf{F}, \textsc{hd}(\underset{(\mu)}{\boxed{(\mu a.\lambda b.(\textsc{if}(b=0,\mathsf{T},\textsc{tl}(a(b-1)))) : \textsc{if}(b=0,\mathsf{F},\textsc{hd}(a(b-1))))}}(0-1)))) : \\
\textsc{if}(\underset{(=0\text{-T})}{\boxed{0=0}}, \mathsf{T}, \textsc{tl}(\underset{(\mu)}{\boxed{(\mu a.\lambda b.(\textsc{if}(b=0,\mathsf{T},\textsc{tl}(a(b-1)))) : \textsc{if}(b=0,\mathsf{F},\textsc{hd}(a(b-1))))}}(0-1))) \\
\Downarrow \\
\underset{(\text{if-T})}{\boxed{\begin{array}{l}
(\textsc{if}(\mathsf{T},\mathsf{F},\textsc{hd}((\lambda a.(\textsc{if}(a=0,\mathsf{T},\textsc{tl}((\mu a.\lambda b.(\textsc{if}(b=0,\mathsf{T},\textsc{tl}(a(b-1)))) \\
: \textsc{if}(b=0,\mathsf{F},\textsc{hd}(a(b-1)))))(a-1)))) : \\
\textsc{if}(a=0,\mathsf{F},\textsc{hd}((\mu a.\lambda b.(\textsc{if}(b=0,\mathsf{T},\textsc{tl}(a(b-1)))) : \\
\textsc{if}(b=0,\mathsf{F},\textsc{hd}(a(b-1)))))(a-1))))(0-1))))
\end{array}}} : \\
\underset{(\text{if-T})}{\boxed{\begin{array}{l}
\textsc{if}(\mathsf{T},\mathsf{T},\textsc{tl}((\lambda a.(\textsc{if}(a=0,\mathsf{T},\textsc{tl}((\mu a.\lambda b.(\textsc{if}(b=0,\mathsf{T},\textsc{tl}(a(b-1)))) : \\
\textsc{if}(b=0,\mathsf{F},\textsc{hd}(a(b-1)))))(a-1)))) : \\
\textsc{if}(a=0,\mathsf{F},\textsc{hd}((\mu a.\lambda b.(\textsc{if}(b=0,\mathsf{T},\textsc{tl}(a(b-1)))) : \\
\textsc{if}(b=0,\mathsf{F},\textsc{hd}(a(b-1)))))(a-1))))(0-1)))
\end{array}}} \\
\Downarrow \\
\mathsf{F} : \mathsf{T}
\end{array}
$$

This is the desired result, of course, since 3 is odd, not even.

Finally we show the language as a CRSar before and after explicit substitution in Figure 4.4 and Figure 4.5, respectively.

The first, the CRSa version, just does the same kind of updating as we did for the λ-calculus above, using the new syntactic trick to tie the cyclic loop. The automatically explicified version essentially does 'stepwise' allocation of memory similar to the sharing in the operational semantic description of Purushothaman and Seaman (1992).

4.5 Discussion

This chapter has contributed two developments for higher order rewriting: first we have contributed to the operational understanding of CRS in general, combining the CRS notion with explicit substitution. Second, we have shown how combinatory reduction systems can be used in the description of functional programming languages.

In summary we have argued for the following table of CRS restrictions and extensions (left) in order to put a bound on the corresponding observational operational property (right).

Syntax. PCF with pairs is a CRS simplifying 'programs' restricted to (closed) expressions e inductively defined by

$$\begin{aligned} e \;::=\;& 0 \mid e+1 \mid e-1 \mid (e:e') \mid \textsc{hd}(e) \mid \textsc{tl}(e) \\ \mid\;& \mathsf{T} \mid \mathsf{F} \mid e=0 \mid \textsc{if}(e,e',e'') \\ \mid\;& x \mid \lambda x.e \mid e\,e' \mid \mu x.e \\ \mid\;& e^{a} \end{aligned}$$

where a ranges over addresses.

Reduction.

$$\begin{aligned}
(\lambda x.Z(x))X &\to Z((X)^{a}) && (\beta)\\
(Z)^{\mathsf{V}}X &\to (Z \,[]\, (Z)^{\mathsf{V}})X && (\beta\text{-copy})\\
\mu x.Z(x) &\to (Z((\blacksquare)^{a}))^{a} && (\mu)\\
\textsc{if}(\mathsf{F},X,Y) &\to Y && (\text{if-F})\\
\textsc{if}(\mathsf{T},X,Y) &\to X && (\text{if-T})\\
\textsc{if}((Z)^{\mathsf{V}},X,Y) &\to \textsc{if}(Z \,[]\, (Z)^{\mathsf{V}},X,Y) && (\text{if-copy})\\
0=0 &\to \mathsf{T} && (=0\text{-T})\\
(X+1)=0 &\to \mathsf{F} && (=0\text{-F})\\
(Z)^{\mathsf{V}}=0 &\to (Z \,[]\, (Z)^{\mathsf{V}})=0 && (=0\text{-copy})\\
\textsc{hd}(X:Y) &\to X && (\text{hd})\\
\textsc{hd}((Z)^{\mathsf{V}}) &\to \textsc{hd}(Z \,[]\, (Z)^{\mathsf{V}}) && (\text{hd-copy})\\
\textsc{tl}(X:Y) &\to Y && (\text{tl})\\
\textsc{tl}((Z)^{\mathsf{V}}) &\to \textsc{tl}(Z \,[]\, (Z)^{\mathsf{V}}) && (\text{tl-copy})\\
((X+1)-1) &\to X && (\text{-1})\\
(Z)^{\mathsf{V}}-1 &\to (Z \,[]\, (Z)^{\mathsf{V}})-1 && (\text{-1-copy})
\end{aligned}$$

Figure 4.4: PCF+pairs with cyclic sharing as CRSar.

$$
\begin{array}{rcll}
(\lambda x.Z(x))X & \longrightarrow & \Sigma([x]Z(x),(X)^{a}) & (\beta\text{-x})\\
\mu x.Z(x) & \longrightarrow & (\Sigma([x]Z(x),(\blacksquare)^{a}))^{a} & (\mu\text{-x})\\
(Z)^{X} & \longrightarrow & (Z \mathbin{[\!]} (Z)^{X}) & (\text{copy})\\
\textsc{if}(F,X,Y) & \longrightarrow & Y & (\text{if-F})\\
\textsc{if}(T,X,Y) & \longrightarrow & X & (\text{if-T})\\
0=0 & \longrightarrow & T & (=0\text{-T})\\
(X+1)=0 & \longrightarrow & F & (=0\text{-F})\\
\textsc{hd}((X:Y)) & \longrightarrow & X & (\text{hd})\\
\textsc{tl}((X:Y)) & \longrightarrow & Y & (\text{tl})\\
(X+1)-1 & \longrightarrow & X & (\text{-1})\\
\Sigma([x](Z_1(x))(Z_2(x)),X) & \longrightarrow & \Sigma([x]Z_1(x),X)\Sigma([x]Z_2(x),X) & (\text{x-@-1})\\
\Sigma([xy]Z(x,y),X) & \longrightarrow & [y]\Sigma([x]Z(x,y),X) & (\text{xma-1})\\
\Sigma([x]\lambda(Z(x)),X) & \longrightarrow & \lambda(\Sigma([x]Z(x),X)) & (\text{x-}\lambda\text{-1})\\
\Sigma([x]\mu(Z(x)),X) & \longrightarrow & \mu(\Sigma([x]Z(x),X)) & (\text{x-}\mu\text{-1})\\
\Sigma([x]\textsc{if}(Z_1(x),Z_2(x),Z_3(x)),X) & \longrightarrow & \textsc{if}(\Sigma([x]Z_1(x),X),\Sigma([x]Z_2(x),X),\Sigma([x]Z_3(x),X)) & \\
 & & & (\text{x-}\textsc{if}\text{-1})\\
\Sigma([x]F,X) & \longrightarrow & F & (\text{x-F-1})\\
\Sigma([x]T,X) & \longrightarrow & T & (\text{x-T-1})\\
\Sigma([x](Z_1(x))=(Z_2(x)),X) & \longrightarrow & (\Sigma([x]Z_1(x),X))=(\Sigma([x]Z_2(x),X)) & (\text{x-=-1})\\
\Sigma([x]0,X) & \longrightarrow & 0 & (\text{x-0-1})\\
\Sigma([x](Z_1(x))+(Z_2(x)),X) & \longrightarrow & (\Sigma([x]Z_1(x),X))+(\Sigma([x]Z_2(x),X)) & (\text{x-+-1})\\
\Sigma([x]1,X) & \longrightarrow & 1 & (\text{x-1-1})\\
\Sigma([x]\textsc{hd}((Z(x))),X) & \longrightarrow & \textsc{hd}((\Sigma([x]Z(x),X))) & (\text{x-}\textsc{hd}\text{-1})\\
\Sigma([x](Z_1(x)):Z_2(x),X) & \longrightarrow & (\Sigma([x]Z_1(x),X)):\Sigma([x]Z_2(x),X) & (\text{x-:-1})\\
\Sigma([x]\textsc{tl}((Z(x))),X) & \longrightarrow & \textsc{tl}((\Sigma([x]Z(x),X))) & (\text{x-}\textsc{tl}\text{-1})\\
\Sigma([x](Z_1(x))-(Z_2(x)),X) & \longrightarrow & (\Sigma([x]Z_1(x),X))-(\Sigma([x]Z_2(x),X)) & (\text{x-}-\text{-1})\\
\Sigma([x]x,X) & \longrightarrow & X & (\text{xv-1-1})\\
\Sigma([x]Z,X) & \longrightarrow & Z & (\text{xgc-1-1})
\end{array}
$$

Figure 4.5: PCF+pairs as CRSar after explicification of substitutions.

variant	bounds	section
explicit substitution	rewrite time	4.2
saturation & free variables	matching time	4.2
sharing & recursion	copying space	4.3

Recall that a 'space cost' always incurs at least an equal 'time cost' because allocation takes time.

In conclusion we claim that CRS-based formalisms are suitable for reasoning about functional program evaluation in general, *i.e.*, including higher order functions, with the full convenience and expressive power of reduction systems. This is particularly useful for functional programming languages because these are often implemented in a way that depends critically on the following technology:

1. Read the program and perform *program analysis* to obtain information about it.

2. Use this information to perform *program transformations* that are 'safe' in the sense that they are *correct* and guaranteed to *terminate*.

3. Discard the analysis information!

4. Perform ordinary evaluation of the transformed program.

The net effect of this is that we first perform some *non-standard reductions*, the program transformations, followed by *standard reductions*, the evaluation. Most of the literature on optimising functional programs uses *ad hoc* reasoning when deciding whether a particular program transformation reduction should be done by the compiler, and the time required for the compiler to do the reduction is rarely included in the argument. With a confluent calculus the two aspects – program transformation and reduction – can be investigated in detail in the same framework!

On the other hand, we might also wish to generalise the notion of reduction of functional programming to add any advanced aspect that we can include without extra cost! This opens up further work with this motto.

> " Obtain a notion of 'functional rewriting' that at the same time captures as much of functional programming and CRS reduction as possible, and where it is ensured that a computer can perform every rewrite step in unitary time. "

The bits and pieces of this chapter will hopefully contribute to this.

Chapter 5

Strategies and Abstract Machines

We explain how explicit substitution provides a technique for separating the issues of *primitives* and *reduction strategies*, and show how an explicit substitution calculus equipped with a strategy can be used to derive an abstract machine. We exploit the techniques of all the previous chapters to give formally understandable justifications for the techniques used for *sharing*, *recursion*, and *parallelism*, in "real" implementations of functional programming languages.

It is well known since Landin (1964) how one can obtain from a *deterministic and compositional strategy* (for λ-calculus) a 'machine' that performs *mechanical evaluation* (of λ-terms) according to this strategy. Such a machine is a nice vehicle for experimenting with implementations because it is then sufficiently rigidly defined to serve as the basis for directions that can be executed by a computer. The problem with this, of course, is that abstract machines are virtually useless for discussing 'clever' program manipulations because these most often correspond to 'rewriting' the program to an equivalent one, in the operational semantics, but in the abstract machine structure the new program seems to have no relation to the original.

The new thing that we add is to show how the *complexity issues of the explicit calculi of the previous chapters are maintained in abstract machines*. This extends the usefulness of such machines because it makes it possible to understand how much a program transformation 'can' change the complexity of a program with respect to the issues of evaluation time and space needed on the abstract machine which is close to what the computer can do. Furthermore, the proximity of the notions means that it is possible to include the cost of program transformation in the overall cost of the evaluation. The way we do this is by presenting *abstract machine derivations* that are sufficiently systematic that it is apparent how the complexity is preserved when we start from *explicit calculi* from the previous chapter where the complexity is already realistic.

In Section 5.1 we recall what is meant by "strategy" or "reduction in context" rules that serve to make reduction of λ-terms deterministic, in particular we discuss *weak* reduction used by essentially all implementations. We next proceed with a presentation of the calculus we will use as the basis for our derivation, $\lambda\sigma^a_w$, which is a calculus with explicit substitution, binding, and addresses restricted to *weak reduction* as is used in all real implementations of programming language we know (corresponding to the lack of runtime code optimisations).

Except for the first section, the material in this chapter is essentially a reproduction of the results reported by Benaissa, Lescanne and Rose (1996) revised to fit with the notations used in these notes.

5.1 Strategies for λ-Calculus

When we wish to actually perform λ-reduction, we are interested in a mechanical *strategy* that gives the normal form (or whichever form we want). The origin of strategies is in 'λ-theories' where they correspond to derivation rules.

This section describes the *evaluation strategies* most commonly used in connection with λ-calculus. We will only describe 'reasonable' strategies in the sense that they are all compositional restrictions of the full λ-calculus (this rules out perpetual strategies, for example).

Thus in general strategies are *restrictions* of the $\xrightarrow[\beta]{}$ reduction introduced in section A.3. Specifically we will explain the 'weak' class of strategies, in particular, the *Call-by-Value* (CBV) strategy used by the functional subsets of the LISP (McCarthy, Abrahams, Edwards, Hart and Levin 1965) and SML (Milner, Tofte and Harper 1990) programming languages, and *Call-by-Name* (CBN) as used by pure functional languages, and finally briefly discuss their 'CPS' (continuation-passing style) intersection.

The possible reduction steps with the various strategies we describe have been illustrated in Figure 5.1; the mentioned strategies are summarised in turn below. They are all constructed from a combination of the following inference rules (and a restriction on the terms in the case of CPS reduction).

5.1.1 Definition (Λ-inferences). Contextual λ-calculus can be expressed as a compositional relation by including the following inference rules (rule names from

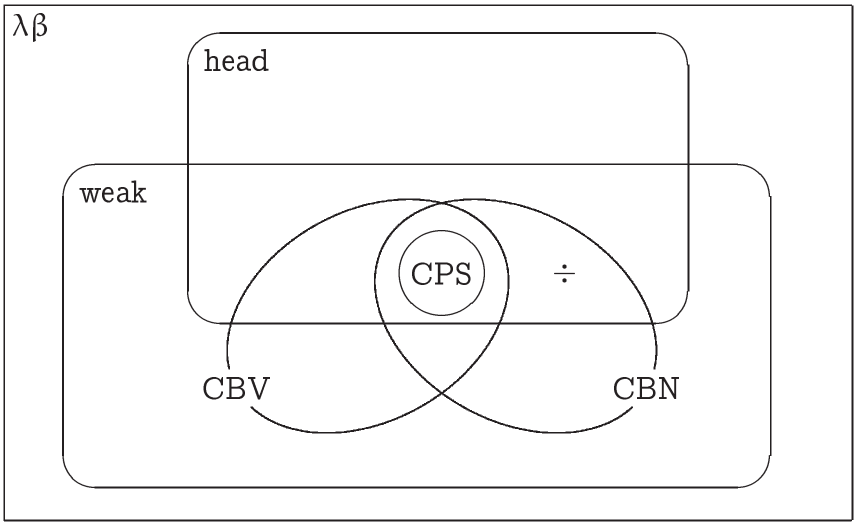


Figure 5.1: Common λ-calculus strategies.

Curry and Feys 1958):

$$\frac{N \to N'}{M\,N \to M\,N'} \qquad (\mu)$$

$$\frac{M \to M'}{M\,N \to M'\,N} \qquad (\nu)$$

$$\frac{M \to M'}{\lambda x.M \to \lambda x.M'} \qquad (\xi)$$

Using these rules we can express λβ-reduction as the '$\beta\mu\nu\xi$-calculus' because it combines the (β) *axiom* with the (μ), (ν), and (ξ) *inference rules* (also called *structural* or *reduction in context* rules).

One can say that this is the 'trivial' strategy in that reduction is allowed in any context. On the other end of the scale the *flat* β*-reduction* which includes no inference rules, is maximally restricted and hence not particularly interesting (but definitely useful).

The various reduction strategies that we will express below will basically be a compromise between these: a subset of the inference rules, sometimes with additional restrictions (that respect the compositionality requirement).

5.1.2 History ("β-theories"). Traditionally inferences are used to define *all* the desired properties of reduction. Such a 'total' description is called a *theory*, and the 'β-conversion theory' is the (β) axiom, the (ν), (μ), and (ξ) rules, and the following additional *closure rules* that realise the reflexive, symmetric, and transitive properties of β-conversion (rule names from Curry and Feys 1958):

$$M \to M \tag{ρ}$$

$$\frac{N \to M}{M \to N} \tag{σ}$$

$$\frac{M \to N \quad N \to P}{M \to P} \tag{τ}$$

We will not use these rules in this tutorial since it is much more convenient to work with 'stepwise' reductions (sometimes known as *small-step semantics* in contrast to *big-step semantics*). There is a similar 'βη-conversion theory' including (η) which we will not discuss either.

It turns out that all the strategies that have been invented for evaluation of λ-terms are *weak reduction strategies*: they are restricted to not include (ξ). This has the property that *any* abstraction is a normal form. Together with the notion of 'head reduction' this gives a coarse classification of reductions.

5.1.3 Definitions (whnf).

1. Weak λβ-reduction, $\xrightarrow[\beta w]{}$, is the relation obtained by omitting (ξ).
2. Head λβ-reduction, $\xrightarrow[\beta h]{}$, is the relation obtained by omitting (μ).

The normal forms of the intersection, omitting (μ) and (ξ), are said to be in *weak head normal form*, abbreviated whnf, and are easily seen to be of the form

$$\text{whnf} ::= \lambda x.M \mid x\,M_1 \ldots M_n$$

for any λ-terms $M, M_1, \ldots, M_n$, $n \in \mathbb{N}_0$ (notice that the second form associates to the left, *i.e.*, is really $(\ldots((x\,M_1)\,M_2)\ldots)\,M_n)$.

In general we will be sloppy and call any subreduction of $\xrightarrow[\beta w]{}$ for 'a weak reduction' and ay subreduction of $\xrightarrow[\beta h]{}$ for 'a head reduction'. Unfortunately the unrestricted weak reduction is not confluent since redexes can be 'captured' underneath a λ:

5.1.4 Observation. $\xrightarrow[\beta w]{}$ *is not CR.*

Proof.

$$\begin{tikzcd}
& (\lambda xy.x)((\lambda x.x)z) \arrow[dl] \arrow[dr] & \\
\lambda y.(\lambda x.x)z \arrow[dr, dotted, two heads, "/" marking] & & (\lambda xy.x)z \arrow[dl] \\
& \lambda y.z &
\end{tikzcd} .$$

□

Hence some further restrictions are necessary to get a manageable system with weak reduction. In practice deterministic reduction is used, and the two strategies we discuss here are both deterministic, weak strategies. They are both defined in the seminal paper of Plotkin (1975); our presentation uses our notation and omits constants.

5.1.5 Definitions (Call-by-? strategies).

1. *Call-by-Value* (CBV) reduction, $\underset{V}{\to}$, is defined as the restriction of the $\lambda\beta$-calculus to the modified axiom

$$(\lambda x.M)N \to M[x := N] \quad \text{if } N \text{ is a value} \qquad (\beta_V)$$

   where "a value" simply means a variable x or an abstraction $\lambda x.M$, and the inferences

$$\frac{N \to N'}{MN \to MN'} \quad \text{if } M \text{ is a value} \qquad (\mu_V)$$

   and (ν).

2. *Call-by-Name* (CBN) reduction, $\underset{N}{\to}$, is defined as the restriction of the $\lambda\beta$-calculus to the (original) axiom (β) and the inferences

$$\frac{N \to N'}{xN \to xN'} \qquad (\mu_N)$$

   and (ν).

These are clearly compositional and deterministic (the side condition of (μ_V) is not a problem because only two kinds of λ-terms are values, so a simple duplication could eliminate the side condition). They are also weak reductions as neither include (ξ) in any form. It turns out that CBN implements $\lambda\beta$ fully whereas CBV is not even normalising:

5.1.6 Propositions. 1. $\underset{V}{\rightarrow}$ *not* PWN of $\underset{\beta}{\rightarrow}$. 2. $\underset{N}{\rightarrow}$ PWN of $\underset{\beta}{\rightarrow}$.

Proofs. 1 is easy by considering $KI\Omega \underset{V}{\not\twoheadrightarrow} I$ but $KI\Omega \underset{V}{\rightarrow} KI\Omega \underset{V}{\rightarrow} \cdots$. 2 is the *standardisation* theorem for closed terms (Plotkin 1975, Theorem 5.1). □

5.1.7 History. Plotkin defines not only the above cited strategies (which are there called 'left reductions' because of the inclusion of (ν) but also two calculi:

1. "λ_V" is $\lambda\beta$ except (β) is replaced by (β_V).

2. "λ_N" is a synonym for $\lambda\beta$.

These calculi are *not* weak and serve to show that the corresponding strategies are 'standard reduction strategies' for something.

Finally, CPS reduction: Plotkin's observation was that terms of the following shape reduce the same with $\underset{V}{\rightarrow}$ and $\underset{N}{\rightarrow}$ (Steele 1978, the CPS name and present formulation origins with).

5.1.8 Definition (continuation-passing style). Let *continuation-passing style* (CPS) terms be defined by

$$M ::= x \mid \lambda x.M \mid Mx \mid M(\lambda x.M)$$

This subset of Λ is denoted Λ_{CPS}.

A more precise view of CPS terms is given by Danvy and Pfenning (1995).

5.1.9 Propositions. For all $M \in \Lambda_{CPS}$,

1. $M \underset{V}{\rightarrow} N$ implies $M \underset{N}{\rightarrow} N$, and $N \in \Lambda_{CPS}$, and

2. $M \underset{N}{\rightarrow} N$ implies $M \underset{N}{\rightarrow} V$, and $N \in \Lambda_{CPS}$.

Proof. Easy case analysis; a side effect is to establish that the area marked with "÷" in Figure 5.1 ($(\mathrm{CBN} \cap \mathrm{head})\backslash\mathrm{CBV}$) is empty. □

5.1.10 History. Abramsky's (1990) *lazy*[1] calculus, $\lambda\ell$, is sometimes claimed as the basis of implementations. $\lambda\ell$-reduction is the relation $\Downarrow_\ell$ inductively defined on closed λ-terms by

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

This is easily seen to be just a 'big-step' version of Plotkin's calculus of Definition 5.1.5.2, restricted to closed terms, *i.e.*,

$$\underset{N}{\twoheadrightarrow}\Big|_{\Lambda^\circ} = \Downarrow_\ell$$

(The proof is by isomorphism between $\underset{N}{\twoheadrightarrow}$-reduction sequences and $\Downarrow_\ell$-derivation trees.) We will not discuss it further, since the definition does not readily show the reduction strategy: it only defines reduction to normal form so the reduction strategy is present only implicitly as a 'proof construction strategy' which is fine except it is not so easy to reason about the relation to other strategies.

5.2 Calculi for Weak Reduction with Sharing

In this section we generalize the weak explicit substitution and binding calculus $\lambda\sigma_w$ defined by Curien, Hardin and Lévy (1996) to include addresses of the kind discussed in the previous chapter.[2]

Figure 5.2 defines syntax and reduction rules of $\lambda\sigma_w^a$. 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 Chapter 1,

[1] The tag 'lazy' is rather misleading: $\lambda\ell$ 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" which is just the same notion as above. Clearly the only closed terms in whnf are abstractions, hence the definition is the same! We will see in section 3.2, however, that these definitions are equivalent to usual lazy evaluation when sharing is included.

[2] The $\lambda\sigma_w$ described here is different from the $\lambda\sigma_w$ of Hardin, Maranget and Pagano (1996).

Syntax. The *addressed preterms* are defined inductively by

$$T, U, V ::= E^a \mid \bot \qquad \text{(Addressed)}$$
$$E, F ::= M\langle s\rangle \mid UV \qquad \text{(Evaluation Context)}$$
$$M, N ::= \lambda M \mid MN \mid \underline{n} \qquad \text{(Pure)}$$
$$s, t ::= \mathsf{id} \mid U \cdot s \qquad \text{(Substitution)}$$

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

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

$$((\lambda M)\langle s\rangle^b\ U)^a \to M\langle U \cdot s\rangle^a \qquad \text{(B)}$$

Weak substitution elimination prereduction. $\twoheadrightarrow\!\!\!\!\!+$ is defined by the rules

$$(MN)\langle s\rangle^a \to (M\langle s\rangle^b\ N\langle s\rangle^c)^a \qquad b, c \text{ fresh} \qquad \text{(App)}$$
$$\underline{0}\langle E^b \cdot s\rangle^a \to E^b \qquad \text{(FVarG)}$$
$$\underline{0}\langle E^b \cdot s\rangle^a \to E^a \qquad \text{(FVarE)}$$
$$\underline{n+1}\langle U \cdot s\rangle^a \to \underline{n}\langle s\rangle^a \qquad \text{(RVar)}$$

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

$$M\langle s\rangle^a \to M\langle s|_{\mathrm{fv}(M)}\rangle^a \qquad s \neq s|_{\mathrm{fv}(M)} \qquad \text{(Collect)}$$

where *environment trimming* is defined by $s|_I = s|_I^0$ where $\mathsf{id}|_I^i = \mathsf{id}$ 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 5.2: $\lambda\sigma_w^a$: syntax and reduction rules.

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 5.5, we study this phenomenon well known for abstract machines as the *space leak problem*.

Furthermore, notice that the rules (FVarG) and (FVarE) have the same LHS: the difference between these two rules is in the choice of the address in the RHS which is either b (FVarG) or a (FVarE). This conceptual choice has a direct correspondence to the duplication versus sharing problem of implementations: if the address is b, the RHS is a redirection of the pointer which refers to the address a (where the term $\underline{0}\langle E \cdot S\rangle$ is) to the address b (where the term E is). As a consequence, further (parallel) rewriting of this argument will be shared with all its other occurrences in the term. If the address is a, a copy of the term E at address b (or, to be precise, a copy of its root node because addresses of its subterms are not changed) is performed. Then the term $\underline{0}\langle E \cdot S\rangle$ at address a is replaced by that copy and later rewriting of this argument will not be shared. Thus we see that the reduction $\twoheadrightarrow\!\!\!\!\!+$ contains two substitution elimination subreductions, $\xrightarrow[gc]{}$ and $\xrightarrow[e]{}$. Let $\sigma_0 = \{(\mathrm{App}), (\mathrm{RVar}), (\mathrm{VarId})\}$, $\sigma g = \sigma_0 \cup \{(\mathrm{FVarG})\}$, and $\sigma e = \sigma_0 \cup \{(\mathrm{FVarE})\}$. 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 descendant redexes (residuals) (Barendregt 1984, Klop 1992) 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 5.2.5.

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

5.2.1 Lemma. Complete developments are finite.

Proof. Same proof as for λ-calculus (Barendregt 1984, p.290). □

The nondeterministic choice between (FVarE) and (FVarG) makes complete development nondeterministic as illustrated by the following example (where we omit id and $\langle \mathsf{id} \rangle$ for clarity):

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

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 after a development. The use of (FVarG) increases sharing whenever (FVarE) decreases it. 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) that the lower three are essentially the same term.

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

Proof. If i is the number of (FVarG/E) redexes in T, then clearly $\#\mathrm{dev}(T) \leq 2^i$. □

5.2.3 Definition (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 $M\langle T_0 \cdots T_n\rangle^a \trianglerighteq_\theta M\langle T_0' \cdots T_n'\rangle^{\theta(a)}$,
- $n^a \trianglerighteq_\theta n^{\theta(a)}$, 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* to U, $T \trianglerighteq U$ if there exists a map θ such that $U \trianglerighteq_\theta (T)$.

5.2.4 Lemma. Let T be a preterm. $(\mathrm{dev}(T), \trianglerighteq)$ is a finite 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. □

Now we can define the parallel extension $\xrightarrow[R]{}$ of a rewrite system R which corresponds intuitively to simultaneous (also known as *Gross-Knuth*) reduction of any number of R-redexes of a term T, *e.g.*, one $\xrightarrow[b]{\#}$ step is the simultaneous reduction of all B-redexes.

5.2.5 Definition (parallel extension). Let T and U be two terms. Then the parallel extension $\twoheadrightarrow\!\!\!\!\!\!\#\;$ of the rewrite relation $\rightarrow$ is defined by $T \twoheadrightarrow\!\!\!\!\!\!\#\; U$ 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.

5.2.6 Definitions (addressing). Given some set of terms with a notion of *address* associated to some subterms. 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 addr(X).
2. The *outermost* a*-addressed subterms* of t is the set $t|a = \{t_1^a, \ldots, t_n^a\}$ with t_i those subterms of t for which an n-ary context $C\{\ldots\}$ exists such that $t = C\{t_1^a, \ldots, t_n^a\}$ and $a \notin \mathrm{addr}(C\{\})$.
3. t is *admissible* if for all $a \in \mathrm{addr}(t)$, $t|a$ is a singleton $\{t'\}$ where $a \notin \mathrm{addr}(t')$.
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 generalizes to any set of addresses $A = \{a_1, \ldots, a_n\}$: $\xrightarrow{A} = \bigcup_{a \in A} \xrightarrow{a}$.
5. $\xrightarrow{\#a}$ (resp. $\xrightarrow{\#A}$) is the parallel extension of $\xrightarrow{a}$ (resp. $\xrightarrow{A}$).
6. Parallel $\rightarrow$-reduction is $\twoheadrightarrow\!\!\!\!\!\!\#\; = \bigcup_{A \subseteq \mathbf{A}} \xrightarrow{\#A}$.

Parallel reduction $\twoheadrightarrow\!\!\!\!\!\!\#\;$ 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 $a \notin \mathrm{addr}(U)$ or $A \cap \mathrm{addr}(U) = \varnothing$ the reductions degenerate: $\{T \mid U \xrightarrow{a} T\} = \{T \mid U \xrightarrow{\#a} T\} = \{T \mid U \xrightarrow{A} T\} = \{T \mid U \xrightarrow{\#A} T\} = \varnothing$.

5.2.7 Proposition. $\xrightarrow[B+\sigma+C]{\#}$ preserves admissibility.

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

5.2.8 Definitions ($\lambda\sigma_w^a$). Given the preterms and prereductions of Figure 5.2.

1. The $\lambda\sigma_w^a$-*terms* are the admissible $\lambda\sigma_w^a$-preterms.

2. $\lambda\sigma_w^a$-*substitution* is $\twoheadrightarrow\!\!\!\!\!\!\shortmid\ \ $,

3. $\lambda\sigma_w^a$-*reduction* is $\xrightarrow[B+\sigma+C]{}$ (or just $\twoheadrightarrow$ when confusion is unlikely).

As for $\lambda\sigma_w$ one associates with the pure term M the $\lambda\sigma_w^a$-term $M\langle\mathsf{id}\rangle^a$ which will then reduce to weak normal form modulo substitution under abstraction. Normal forms V are of the form $(\lambda M)\langle V_1\cdots V_n\rangle$ or $\underline{n}V_1\cdot V_m$. In the rest of the paper, we will only consider *closed* weak normal forms, called *values*, *i.e.*, $(\lambda M)\langle V_1\cdots V_n\rangle$, supposing that only closed terms are evaluated.

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 $\lambda\sigma_w^a$. Both lemmas use the translation function "erase" which deletes all addresses of an addressed term, obviously resulting in a $\lambda\sigma_w$-term.

5.2.9 Lemma (projection). Let T and U be two addressed terms. If $T\twoheadrightarrow U$ then $\mathsf{erase}(T)\xrightarrow[\lambda\sigma_w]{}\!\!\!\!\!\rightarrow \mathsf{erase}(U)$.

Proof. Easy induction on the structure of T.

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

$$\mathsf{erase}(T) = M_1M_2\langle\mathsf{erase}(s)\rangle \xrightarrow[\lambda\sigma_w]{} M_1\langle\mathsf{erase}(s)\rangle M_2\langle\mathsf{erase}(s)\rangle = \mathsf{erase}(U)$$

  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 rule can be applied. if $a \neq b$ then the induction hypothesis applies.

$\square$

5.2.10 Lemma. 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 $W = \mathsf{erase}(T)$, $W' = \mathsf{erase}(U)$, and $T\twoheadrightarrow U$.

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.

$$\begin{aligned}
\mathsf{label}(R_1R_2)_p &= (\mathsf{label}(R_1)_{1p}\mathsf{label}(R_2)_{2p})^p \\
\mathsf{label}(M\langle R_1 \cdots R_n \cdot \mathsf{id}\rangle)_p &= M\langle \mathsf{label}(R_1)_{1p} \cdots \mathsf{label}(R_n)_{np} \cdot \mathsf{id}\rangle^p \\
\mathsf{label}(\underline{n}_p) &= \underline{n}^p
\end{aligned}$$

Then it is easy to show that $\mathsf{label}(M) \xrightarrow{a}\!\!\!\!\!\!\!+\;\; \mathsf{label}(N)$. □

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

5.3 Reduction Strategies

In this section we show how conventional weak reduction strategies can be described through restrictions of the $\lambda\sigma^a_w$-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.

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.

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

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

Strategy	(FVar?)	(Scl)	(Par)	(Sub)	(Hd)	(Tl)	(Sap)	(Lap)	(Rap)
✓	E or G	✓	✓	✓	✓	✓	✓	✓	✓
CBN	E	✓	×	×			①	¬①	×
CBV	E	✓	×	×			②	¬①	③
CBNeedE	E	④	×	¬④	✓	×	①	¬①	×
CBNeedG	G	✓	×	×			①	¬①	×
Specul-$\|\|_n$	E	✓	⑤	✓	✓	✓	✓	✓	✓
Spine-$\|\|_n$	E	✓	⑤	✓	✓	✓	✓	✓	×

✓: "always applicable;" ×: "never applicable;" blank: "don't care;" ①: U is a value; ②: U and V are values; ③: U is a value and V is not; ④: if $s = U \cdot s'$ and $M = \underline{0}$ then U is a value; and ⑤: $\#A \leq n$.

Figure 5.3: Strategies for $\lambda\sigma_w^a$.

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

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

$$\frac{}{M\langle s\rangle^a \vdash \{a\}}\ (\mathrm{Scl}) \qquad \frac{U \vdash A_1 \quad U \vdash A_2}{U \vdash A_1 \cup A_2}\ (\mathrm{Par})$$

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

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

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 $\vdash_{\mathcal{S}}$. 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 $(\lambda M)\langle s\rangle^a$.

Figure 5.3 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 nondeterminism. 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 mentioned in Section 5.2 (and here permitted only in the first strategy) is the choice between (FVarE) and (FVarG).

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

5.3.3 Proposition. $\xrightarrow[X/\checkmark]{\|} = \xrightarrow[X]{\|}$.

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

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). In fact, they uses (FVarE) which does not share 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, Felleisen, Maraist, Odersky and Wadler 1995); CBNeedG is an adaption of Wadsworth's (1971) "λ-graph reduction" to weak reduction.

The last two strategies realize n-*parallelism with sharing*. The *Speculative*-$\|_n$ 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. Our intuition about such reduction strategies is that they only reduce 'needed' redexes in a certain sense. This is captured by the following:

5.3.4 Definition (head terms). Closure Terms are admissible closed addressed terms and defined inductively by

$$P ::= M\langle s_h\rangle^a \mid (PM\langle s_h\rangle^a)^b \qquad \text{such that } a \notin \mathsf{addr}(P_l) \qquad \text{(Head Term)}$$

$$s_h ::= P \cdot s_h \mid \bot \cdot s_h \mid \mathsf{id} \qquad \text{(LSubstitution)}$$

We call terms of the form $\_\langle\_\rangle$ *closures*.

Head addresses The set $\mathcal{H}(P)$ of head addresses of a closure term P is defined by the following inference rules.

$$\frac{s \vdash A}{M\langle s\rangle^a \vdash \{a\} \cup A} \qquad \frac{P \vdash A_1 \quad s \vdash A_2}{(PM\langle s\rangle^b)^a \vdash \{a\} \cup A_1 \cup A_2} \qquad \frac{P \vdash A_1 \quad s \vdash A_2}{P \cdot s \vdash A_1 \cup A_2} \qquad \frac{}{\mathsf{id} \vdash \{\}}$$

Queue addresses The set $\mathcal{Q}(P)$ of queue addresses of a closure term P is defined by the following inference rules.

$$\frac{s \vdash A}{M\langle s\rangle^a \vdash A} \qquad \frac{P \vdash A_1 \quad s \vdash A_2}{(PM\langle s\rangle^b)^a \vdash \{b\} \cup A_1 \cup A_2} \qquad \frac{P \vdash A_1 \quad s \vdash A_2}{P \cdot s \vdash A_1 \cup A_2} \qquad \frac{}{\mathsf{id} \vdash \{\}}$$

Head Terms we say that P is a *Head term* if it is a closure term and $\mathcal{H}(P) \cap \mathcal{Q}(P) = \{\}$.

We will study these strategies. We start by focusing on the structural properties of terms preserved by these strategies. Then, we narrow our attention to just 'call-by-need-like' strategies.

5.3.5 Proposition. Let $M\langle \mathsf{id}\rangle^a \not\twoheadrightarrow_{\mathcal{S}} T$. Then

1. If $\mathcal{S}$ disables (Rap) and enables (FVarE) only when E (in $\underline{0}\langle E^b \cdot s\rangle^a$) is a value then T is of the form $(\cdots(N_0\langle s_0\rangle^{a_0}\ N_1\langle s_1\rangle^{a_1})^{b_1} \cdots N_n\langle s_n\rangle^{a_n})^{b_n}$ where $n \geq 0$.

2. If $\mathcal{S}$ disables both (FVarG), (Sub), and (Rap) then all substitutions contained in T are of the form $\langle N_1\langle s_1\rangle^{a_1} \cdot \ldots \cdot N_n\langle s_n\rangle^{a_n}\rangle$.

Proof. Given T and $a, b \in \mathsf{addr}(T)$, we say that b is the *right-subaddress* of a if b occurs only in contexts $T = C[(U\ E^b)^a\}$, and we say that T is *argument local* if all addresses b of such contexts are right-subaddresses of the correponding a. Since the freshness addresses condition of (App), and (FVarE) is disabled for terms of the form $\underline{0}\langle(T\ U)^b \cdot s\rangle^a$, then rewriting preserves the argument local property. If we also disallow (Rap) then an argument local term has exactly the form of the first assertion which thus holds. The second assertion is easily checked by analysis of the possible rewrite steps. $\square$

Figure 3.2 showed Ariola et al.'s (1995, Figure 3) "standard call-by-need λ_{let}-calculus." In λ_{let} 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} must correspond to $\lambda\sigma_w^a$ with an environment-based evaluation strategy.

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

Proof. The proof is based on constructing translations between the two calculi, proving the soundness and completeness diagrams

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

A simple analysis shows that each reduction step in one system is reproducible as (possibly several) steps in the other, using the above translations and Proposition 5.3.5 to permit disregarding non-argument local terms.

To show the equivalence between our call by need and this 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.

5.3.7 Definition (translation λ_{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 $\mathcal{S}[\![M]\!]\,\mathsf{id}\,()$ given by

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

with a and b fresh everywhere.

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

$$\begin{aligned} \mathcal{A}(M, \{E^b\} \cup \mathbf{S}) &= \mathcal{A}(\mathsf{let}\ b = M'\ \mathsf{in}\ M, \mathbf{S}) \quad \text{where } (M', \mathbf{S}') = \mathcal{C}[\![E]\!]\,\mathbf{S} \\ \mathcal{A}(M, \varnothing) &= M \\ \mathcal{C}\big[\!\big[E^a M\langle s\rangle^b\big]\!\big]\,\mathbf{S} &= (\ M'\,(\mathcal{C}[\![M\langle s\rangle]\!]\,())\ ,\ \mathbf{S}'\ ) \quad \text{where } (M', \mathbf{S}') = \mathcal{C}[\![E]\!]\,(\mathbf{S} \cup \{s\}) \\ \mathcal{C}[\![M\langle s\rangle]\!]\,\mathbf{S} &= (\ \mathcal{V}[\![M\langle s\rangle]\!]\,()\ ,\ \mathbf{S} \cup \{s\}\ ) \end{aligned}$$

and $\mathcal{V}$ assigns names to namefree terms: $\mathcal{V}\big[\!\big[M\langle E^b \cdot s\rangle\big]\!\big]\rho = \mathcal{V}[\![M\langle s\rangle]\!](\rho + b)$; $\mathcal{V}[\![M\langle\mathsf{id}\rangle]\!]\rho = \mathcal{V}[\![M]\!]\rho$; $\mathcal{V}[\![\lambda M]\!]\rho = \lambda x.\mathcal{V}[\![M]\!](y\cdot\rho)$ with y a fresh variable; $\mathcal{V}[\![MN]\!]\rho = (\mathcal{V}[\![M]\!]\rho)\,(\mathcal{V}[\![N]\!]\rho)$; and $\mathcal{V}[\![\underline{n}]\!]\,\rho = \rho(n)$.

From these the diagram above follows, hence CBNeedE is a correct and complete model of Ariola et al.'s (1995) call-by-need. □

5.4 Constructors and Recursion

In this section we deal with two important features of pure 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.

5.4.1 Definition (simple data structures). $\lambda\sigma c^a_w$ is the system obtained by extending the definition of $\lambda\sigma^a_w$ ("pure") preterms to include

$$M, N ::= \cdots \mid C_{i,j} \mid \langle\cdots; C_{i,j_i} : M_i; \cdots\rangle$$

where $C_{i,j}$ is a *constructor* with "identifier" i and rank j which means that in the selection component $C_{i,j} : M_i$, M_i binds the indices $\underline{0} \ldots \underline{j-1}$ in M_i. Data values, possibly with arguments, can (only) be used to select one of these by the reduction

$$(C_{i,j}\langle T_0 \cdots T_{j-1} \cdot s\rangle^b \ \langle\cdots; C_{i,j} : M_i; \cdots\rangle\langle t\rangle^c)^a \to M_i\langle T_0 \cdots T_{j-1} \cdot t\rangle^a \qquad \text{(Case)}$$

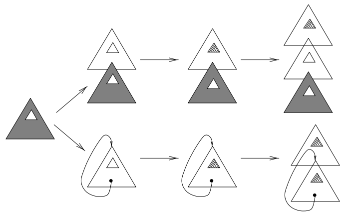
In order for constructors to behave as functions in programs and "consume" the appropriate number of arguments they should only be used inside the right number of abstractions, *i.e.*, we can encode lists as $\mathsf{Nil} \equiv C_{0,0}$ and $\mathsf{Cons} \equiv \lambda\lambda C_{1,2}$.

For recursion denoted by μM, where μ binds index $\underline{0}$ in M, we have two alternatives. $\lambda\sigma c\mu^a_w$ is what is obtained with the easy solution, which is to include the rule

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

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

Another solution consists in encoding unfolding such that it is always delayed 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 and Klop 1994). We have chosen to do this using the $\bullet^a$ "backpointer" syntax (Felleisen and Friedman 1989, Rose 1996b): reducing a fixed point operator places a $\bullet$ at the location where unfolding should happen when (if) it is needed.



The difference is illustrated to the right. Consider the initial term with a 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 we pay is that some potential redexes are lost, since backpointers are syntactic and thus can block specific redexes. Also, the admissibility definition becomes slightly more complicated.

5.4.2 Definition (cyclic addressing). Given a set of addressed terms $\mathbf{T}$.

1. The *cyclic addressed preterms*, $\mathbf{T}^\bullet$, allows subterms of the form $\bullet^a$. Preterms without $\bullet$ are called *acyclic*.

2. t is *cyclic admissible* if all addresses $a \in \operatorname{addr}(t)$ satisfy either $t|a = \{\bullet^a\}$, or $t|a = \{F^a\}$ and $F|a \subseteq \{\bullet^a\}$. As before admissible preterms are called *terms*.

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

The cyclic extension of a set of rules is thus derived by inserting explicit unfolding *where an address is removed.*

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

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

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

$$\mu M\langle s\rangle^a \rightarrow M\langle \bullet^a \cdot s\rangle^a \qquad \text{(Cycle)}$$

Let us briefly consider what the "cyclic extension" means. It is clear that (Cycle) preserves (cyclic) admissibility but not all 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 Figure 5.2 as follows:

$$((\lambda M)\langle s\rangle^b\, U)^a \to M\langle U \cdot (s[\bullet^b := \lambda M\langle s\rangle^b])\rangle^a \tag{B$_w^\bullet$}$$
$$\underline{0}\langle E^b \cdot s\rangle^a \to (E[\bullet^a := \underline{0}\langle E^b \cdot s\rangle^a])^b \tag{FvarG$^\bullet$}$$
$$\underline{0}\langle E^b \cdot s\rangle^a \to (E[\bullet^b := E^b])^a \tag{FvarE$^\bullet$}$$

and (Case) from above changes as follows:

$$(C_{i,j}\langle T_0 \cdots T_{j-1} \cdot s\rangle^b\ \langle\cdots; C_{i,j} : M_i; \cdots\rangle\langle t\rangle^c)^a \to M_i\langle T_0' \cdots T_{j-1}' \cdot t'\rangle^a \tag{Case$^\bullet$}$$

where $T_i' = T_i[\bullet^b := C_{i,j}\langle T_0 \cdots T_{j-1} \cdot s\rangle^b]$ and $t' = t[\bullet^c := \langle\cdots; C_{i,j} : M_i; \cdots\rangle\langle t\rangle^c]$.

5.4.4 Theorem. *$\lambda\sigma c\mu_w^{a\bullet}$ is correct w.r.t. $\lambda\sigma c\mu_w^a$.*

Proof. The difficult point is to prove that delaying unfolding is safe. We do this by considering $\lambda\sigma c\mu_w^{a\bullet}\rho$ *i.e.*, $\lambda\sigma c\mu_w^{a\bullet}$ annotated with the history of cyclic pointers $\bullet$. The rule (Cycle) becomes $\mu M\langle s\rangle_\rho^a \to M\langle\bullet^a \cdot s\rangle^a_{[\bullet := \mu M\langle s\rangle^a \rho, \rho]}$, whereas the other rules distribute the history into the term. Then we can show that if $M_\rho \xrightarrow[\lambda\sigma c\mu_w^{a\bullet}\rho]{\Vert\ \bullet} N_\rho$ then $M\rho \xrightarrow[\lambda\sigma c\mu_w^a]{\Vert} N\rho$. $\square$

5.5 A Space Leak Free Calculus

A drawback of calculi such as $\lambda\sigma_w^a$ is that they leak space: not only can $\lambda\sigma_w^a$-terms contain unreachable subterms but these may never be removed by reduction.

For instance, consider the term $((\mu\lambda\underline{1}\underline{1})I)\langle\mathrm{id}\rangle^a$ reduces (with call-by-need) to a term of the form

$$\big(\ (\lambda\underline{1}\underline{1})\langle(\lambda\underline{1}\underline{1})\langle\bullet^b\rangle^b\rangle^d\ \underline{1}\langle\underline{1}\langle\cdots\underline{1}\langle I\langle\mathrm{id}\rangle^c \cdot (\lambda\underline{1}\underline{1})\langle\bullet^b\rangle^b\rangle^{d_n}\cdots\rangle^{d_2}\rangle^{d_1}\ \big)^a$$

with infinitely growing garbage (situations like these do, in fact, happen in real implementations).

In this section, we formalise the notion of garbage free terms and space leak free reduction. Then we present the "Trim" calculus, $\lambda\sigma_w{\downarrow}gc$, which is space leak free for all strategies by ensuring that (Collect) is invoked sufficiently often that no garbage persists, essentially implementing a kind of run-time garbage collection. Finally, we show how the restriction of $\lambda\sigma_w{\downarrow}gc$ to call-by-need reduction degenerates to the simple environment trimming used for this purpose by the STG-machine of Peyton Jones (1992).

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

$$\begin{aligned} T_f &::= M\langle s_f\rangle^a \mid (T_f T_f)^a \mid n^a \mid \bot \quad \text{with } s_f = s_f|_{\mathrm{fv}(M)} \\ s_f &::= \mathsf{id} \mid T_f \cdot s_f \end{aligned}$$

5.5.2 Proposition. If $T \xrightarrow[\text{cp}]{\text{addr}(T)} U$, then U is a garbage-free term.

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

Space leak freeness (first precisely formulated for a lazy abstract machine by Sestoft 1994) means that every (Collect)-redex will eventually disappear in a computation. In other words, no unreachable subterm stays indefinitely.

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

$$D : T_0 \xrightarrow{A_1} T_1 \xrightarrow{A_2} T_2 \xrightarrow{A_3} \cdots$$

D is *space leak free* if $T_n = C[E^b]$ where E^b is garbage (meaning that $\xrightarrow[C]{}$ can be used to replace it with $\bot$) implies that there exists $m\rangle n$ such that E^b is collected in T_m. A reduction relation $\xrightarrow[R]{}$ (of $\lambda\sigma_w^a$) is space leak free if all its reduction paths starting at a garbage-free term are space leak free.

A naïve way to provide space leak reduction consists in alternating between $B+\sigma$ reduction and a complete Collecting as stated in the following proposition.

5.5.4 Proposition. $\xrightarrow[B+\sigma]{} \!\!\!\!\rightarrow \cdot \xrightarrow[\text{cp}]{}$ is space leak free.

Proof. By Proposition 5.5.2, it is clear that after each reduction step, there remain no addressed subterms which are garbage. $\square$

In the rest of this section, we present a calculus, $\lambda\sigma_w\!\downarrow\!\text{gc}$, that never introduces space leaks. The principle consist in analyzing the rules of $\lambda\sigma_w^a$. One notices that only (B), (App), and (Case), can produce unreachable terms. (App) introduces two new subaddressed terms of the form $\_\langle\_\rangle^a$ (closures) on which an application of (Collect) might be necessary. (B) adds a new addressed term (argument) to the function's substitution; if we know that the B-redex is garbage free then it is necessary to check only whether the argument is reachable or not. (Case) can 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 (using the acyclic rules but can easily be generalised the cyclic calculus since neither (Unfold) nor (Cycle) introduces space leaks):

$$(\lambda M\langle s\rangle^{b} T)^{a} \to \begin{cases} M\langle \bot \cdot s\rangle & \text{if } \underline{0} \notin \mathrm{fv}(M) \\ M\langle T \cdot s\rangle & \text{if } \underline{0} \in \mathrm{fv}(M) \end{cases} \tag{TB}$$

$$(MN)\langle s\rangle^{a} \to (M\langle s|_{\mathrm{fv}(M)}\rangle^{b} N\langle s|_{\mathrm{fv}(N)}\rangle^{c})^{a} \quad b, c \text{ fresh} \tag{TApp}$$

$$(C_{i,j}\langle T_0 \cdots T_{j-1} \cdot s\rangle^{b}\ \langle \cdots ; C_{i,j} : M_i ; \cdots\rangle\langle t\rangle^{c})^{a} \to M_i\langle T_0 \cdots T_{j-1} \cdot t|_{\mathrm{fv}(N_i)}\rangle^{a} \tag{TCase}$$

5.5.5 Definition. The $\lambda\sigma_w{\downarrow}$gc-*calculus* is the calculus over addressed terms combining the original (FVarE), (FVarG), (RVar), and (Cycle) with the new (TB), (TApp), and (TCase). We write this reduction $\xrightarrow[\mathrm{T}]{}$.

5.5.6 Remark. Notice that $\xrightarrow[(\mathrm{TApp})]{a} = \xrightarrow[(\mathrm{App})]{a} \cdot \xrightarrow[\mathrm{cp}]{b} \cdot \xrightarrow[\mathrm{cp}]{c}$, and $\xrightarrow[(\mathrm{TCase})]{a} = \xrightarrow[(\mathrm{Case})]{a} \cdot \xrightarrow[\mathrm{cp}]{q, d_1, \ldots d_m}$.

5.5.7 Theorem. *Let* T *be a garbage-free term. If* $T \xrightarrow[\mathrm{T}]{} U$ *then* U *is a garbage-free term.*

Proof. By case analysis of each rule of $\xrightarrow[\mathrm{T}]{}$. □

5.5.8 Corollary. $\lambda\sigma_w{\downarrow}$gc *is space leak free.*

Since we cannot rewrite in unreachable terms, we can claim that $\lambda\sigma_w{\downarrow}$gc is isomorphic modulo substitution under λ to the weak version of Wadsworth's (1971) "λ-graph reduction." $\lambda\sigma_w{\downarrow}$gc ensures that all its strategies are space leak free. However, we remind the reader that collecting incurs an overhead, so it is desirable to minimize this task. One way is to exploit structural properties enforced by the used strategy. In the remainder of this section we study optimization provided by call-by-need w.r.t. space leak freeness, in particular, we show that a state of the art implementation, the STG-machine (Peyton Jones 1992), 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 $C[(MN)\langle s\rangle]$ then we know that the left term ($M\langle s\rangle$) will be evaluated first. Hence trimming this term is unnecessary since it cannot leak space. Similarly, a trimmed version of (B) and (Case) are not necessary. The only remaining rule, (App), becomes

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

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

5.5.9 Theorem. $\lambda\sigma_w{\downarrow}\mathrm{gc}_N$ *is space leak free for call-by-name (CBN) and call-by-need (both CBNeedE and CBNeedG).*

Proof. $\lambda\sigma_w{\downarrow}\mathrm{gc}_N$ trims all terms where computations is forbiden, *i.e.*, right argument of applications. □

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

Proof. In the STG-machine, the arguments of applications and constructors must be variables which are bound by a *let*-expression. Consider the term $MN_1 \ldots N_n$ which in λ_{let} is written $\mathsf{let}\ x_1 = N_1, \ldots x_n = N_n\ \mathsf{in}\ Mx_1 \ldots x_n$ where $N_1, \ldots, N_n$ are annotated with their free variables. The STG machine computes (see Peyton Jones 1992, section 5.3) that expression by placing the N_i's in the heap and trimming their associated environments. This obviously corresponds to n applications of (TAppN). □

5.6 Discussion

We have demonstrated how explicit substitution is a useful framework for reproducing known abstract machine technology in a systematic and reliable way.

The uniform treatment of the various concrete aspects of mechanical evaluation is new. The correctness of the derivations automatically prove results that relate the various explicit dimensions, in particular our presentation highlights *why* the correctness proofs *etc* of previously published abstract machines are so relatively simple. We also consider the generalisation of the *classification system* of the previous chapter to a 'strategy dimension' a contribution.

Appendix A

Preliminaries

This appendix makes these notes self-contained by summarising the concepts which are glued together by explicit substitution: Section A.1 explains the reduction concept of (abstract) rewriting and Section A.2 the inductive notations of semantics. Finally Section A.3 gives a short recollection of the untyped $\lambda\beta$-calculus with special emphasis on the rôle of substitution and variable binding.

A.1 Reductions

Informally, 'reduction' means stepwise simplification of some problem, usually with the intent of eventually reaching a 'solution'. This is also the formal idea which we will explain here, since it makes reasoning about stepwise evaluation as well as many auxiliary things such as orderings, *etc.*, much easier.

Our treatment is based on definitions from the standard literature (Huet 1980, Klop 1992, van Oostrom 1994). A particularly useful technique is the 'stencil diagram' notation of Rosen (1973) to express reduction propositions.

We start by fixing the notation and terminology for relations (we assume the reader is familiar with basic set notations). Then we explain using diagrams for reasoning about relations, and state several useful properties of relations, notably *translation* and *projection*. Finally we define several standard properties such as confluence and normalisation.

A.1.1 Notations (relations).

1. An n-*ary relation* ($n \geq 2$) is a subset $R \subseteq A_1 \times \ldots \times A_n$ where we write $R(a_1, \ldots, a_n)$ for $(a_1, \ldots, a_n) \in R$.

2. For *binary relations* we use 'directed' infix symbols such as $\rightarrow$, write $a \rightarrow b$ for $(a, b) \in \rightarrow$, and say it is a relation *from* A_1 *to* A_2; A_1 is called the *domain* and A_2 the *range*.

3. A binary relation from A to A is called a *reduction on* A.

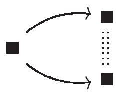
We use ordinary set constructions for relations: $\cap$, $\cup$, $\subseteq$, *etc.*; in particular if $\rightarrow$ is a reduction on A, then the *restriction* of $\rightarrow$ to B, written $\rightarrow|_B$, is defined as $\rightarrow \cap B \times B$, and the *complement* of $\rightarrow$, written $\not\rightarrow$, is defined as $A \times A \backslash \rightarrow$ (sometimes written $\complement_{A \times A} \rightarrow$). Finally, we denote the natural numbers from 1 by $\mathbb{N}$; when 0 is included we write $\mathbb{N}_0$, and we write the empty set as $\varnothing$.

A.1.2 History. We have borrowed the word *reduction* from Huet (1980); the concept is called a *general replacement system* (GRS) by Rosen (1973), an *abstract reduction system* (ARS) by Klop (1992), and an *abstract rewrite system* by van Oostrom (1994).

Most relations in this tutorial will be reductions. In particular we will use the common equality sign $=$ to mean *equality* in the strictest syntactic sense available in each case.[1] If the base set is unclear we write $=|_A$ (one can think of this as 'the universal $=$ relation restricted to A'). We also use $=$ on entire relations, *e.g.*, $\rightarrow = \leftarrow$ is a way to express that $\rightarrow$ is symmetric.

Next we give a mathematical definition of diagram propositions, formalising the description of Rosen (1973), however, it is not diffcult to understand by considering the following example definitions of some well-known standard properties of abstract relations.

A.1.3 Definitions (examples of properties as diagrams).

1. $\rightarrow$ is a *function*, or *deterministic*, if  . A function defined over an enumerable range is a *map*.

2. $\rightarrow$ is *total* if , otherwise it is *partial* (but we will rarely say so).

3. Given $\rightarrow$. The *converse* of $\rightarrow$ is the relation $\rightarrow^{\times}$ defined by  ; it is simply denoted $\leftarrow$.

[1]Thus "$=$" is not used to denote β-convertibility.

The formal definition below translates these as follows: 1 becomes $\forall a, b, c : (a \to b \land a \to c) \implies (a = b)$, 2 becomes $\forall a \, \exists b : a \to b$, and 3 becomes $\forall a, b : (a \to b) \implies (b \not\to a)$. The definition of converse means that we must take care to use asymmetric symbols for asymmetric relations in diagrams.

A.1.4 Notation (diagrammatic propositions). A diagram is a picture with 'solid' and 'dotted' relation 'arrows' placed between 'plain' and 'framed' nodes. Such a picture denotes a relational proposition of the form

$$\forall V^{\blacksquare} : P^{\blacksquare} \text{ implies } \exists V^{\square} : P^{\square}$$

where $V^{\square}$ denotes the (unbound) node variables occuring *only* framed, $P^{\square}$ is a conjunction of all relations where the relation symbol is dotted (*e.g.*, $\dashrightarrow$ and $\dashrightarrow$ are dotted versions of $\to$), and $P^{\blacksquare}$ denotes the conjunction of all the remaining (solid) relations.

Diagram entries can be given as $\blacksquare$ (or $\square$) when they are just meant to be anonymous plain (or framed) variables implicitly distinct from all other variables in the proposition; sometimes we omit framing if it is obvious.

Finally, we allow *diagrammatic reasoning*: suppose we are given a *diagrammatic assertion* which we wish to prove, *i.e.*, the dotted/framed parts are 'unknown'. We can then use another diagram that 'fits' as a lemma such that some of these unknowns can be 'painted' to become solid/plain; if this process terminates with a completely solid/plain diagram then we have proven the original assertion.

A.1.5 Definitions. Let $\to$ be a reduction on A.

1. It is *reflexive* iff $\begin{tikzcd} \blacksquare \arrow[loop right, dotted] \end{tikzcd}$.

2. It is *transitive* iff $\begin{tikzcd} \blacksquare \arrow[r, bend left] \arrow[rr, bend right, dotted] & \blacksquare \arrow[r, bend left] & \blacksquare \end{tikzcd}$.

3. It is *symmetric* iff $\begin{tikzcd} \blacksquare \arrow[r, bend left] & \blacksquare \arrow[l, bend left, dotted] \end{tikzcd}$.

4. It is *antisymmetric* iff $\begin{tikzcd} \blacksquare \arrow[r, bend left] \arrow[r, dotted, no head, equal] & \blacksquare \arrow[l, bend left] \end{tikzcd}$.

5. It is an *equivalence* if it is reflexive, transitive, and symmetric (such a reduction is sometimes called a *conversion*).

6. It is a *partial order* if it is reflexive, transitive, and antisymmetric.

A.1.6 Definition (composition). Let $\xrightarrow[AB]{}$ be a binary relation from A to B, $\xrightarrow[BC]{}$ be from B to C. The *composition* $\xrightarrow[AB]{} \cdot \xrightarrow[BC]{}$ is $\xrightarrow[AC]{}$ defined as the least binary relation satisfying $\begin{tikzcd} \blacksquare \arrow[r, "AB"] \arrow[rr, dotted, bend right, "AC"'] & \blacksquare \arrow[r, "BC"] & \blacksquare \end{tikzcd}$.

In particular $\leftarrow \cdot \rightarrow = (=|_B)$ if $\rightarrow$ is a total function from A to B. We will use several shorthands for reduction compositions.

A.1.7 Notations (compositions). Let $\rightarrow$ be a reduction on A.

1. $\xrightarrow{n}$, for $n \in \mathbb{N}_0$, is the n*-fold composition* defined inductively by

$$\xrightarrow{n} = \begin{cases} =|_A & \text{if } n = 0 \\ \rightarrow \cdot \xrightarrow{n-1} & \text{if } n > 0 \end{cases}$$

2. $\xrightarrow{n:m}$, where $n, m \in \mathbb{N}_0$, is the union $\xrightarrow{n} \cup \xrightarrow{n+1} \cup \cdots \cup \xrightarrow{m}$ (the empty relation if $n > m$). If we omit n and/or m then 0 and/or ∞, respectively, are understood.[2]

Usually one establishes a property by either extending or restricting the relation as little as possible to ensure it.

A.1.8 Notations (closures). Let $\rightarrow$ be a reduction on A. For some concept X, the X *closure* of $\rightarrow$ means the (unique) smallest reduction $\rightsquigarrow$ on A which satisfies X and such that $\rightsquigarrow \supseteq \rightarrow$. In particular,

1. The reflexive and/or transitive closures of $\rightarrow$ are defined by composition: $\xrightarrow{=} = \xrightarrow{:1}$, $\xrightarrow{+} = \xrightarrow{1:}$, and $\twoheadrightarrow = \xrightarrow{:}$. (The first and last are denoted $\xrightarrow{\epsilon}$ and $\xrightarrow{*}$ by Huet (1980).)

2. We will use $\twoheadleftarrow\!\twoheadrightarrow$ for $\rightarrow$'s *equivalence closure*.[3]

A.1.9 Notation (restrictions). Let $\rightarrow$ be a reduction on A. For some concept X, the X *restriction* of $\rightarrow$ means the largest reduction $\rightsquigarrow$ on A which satisfies X and the requirement that $\rightsquigarrow \subseteq \rightarrow$.

The following will be our chief tool for comparing reduction relations.

[2] ∞ just means there is no upper limit (not infinitary reduction or such).

[3] This is *not* always obtainable by constructing the reflexive, transitive and then symmetric closures in some sequence, of course, even if the notation $\twoheadleftarrow\!\twoheadrightarrow$ might be interpreted to indicate just that.

A.1.10 Definitions (translation). Given $\underset{A}{\rightarrow}$ and $\underset{B}{\rightarrow}$ on A and B, respectively, and a relation $\rhd\!\!-$ from A to B.

1. $\rhd\!\!-$ is a *translation* of $\underset{A}{\rightarrow}$ to $\underset{B}{\rightarrow}$ if $\begin{tikzcd} \blacksquare \arrow[r, "A"] \arrow[d, dotted, "\rhd"'] & \blacksquare \arrow[d, dotted, "\rhd"] \\ \square \arrow[r, dotted, "B"'] & \square \end{tikzcd}$ and $\begin{tikzcd} \square \arrow[r, dotted, "A"] \arrow[d, dotted, "\rhd"'] & \square \arrow[d, dotted, "\rhd"] \\ \blacksquare \arrow[r, "B"'] & \blacksquare \end{tikzcd}$.

2. $\rhd\!\!-$ is a *projection* of $\underset{A}{\rightarrow}$ into $\underset{B}{\rightarrow}$ if furthermore $\begin{tikzcd} \blacksquare \arrow[r, "A"] \arrow[d, "\rhd"'] & \blacksquare \arrow[d, "\rhd"] \\ \blacksquare \arrow[r, dotted, "B"'] & \blacksquare \end{tikzcd}$.

A.1.11 Remark. A translation is a *many-to-many relation*, hence the converse of a translation is again a translation. The additional requirement of a projection means that it is a *many-to-one relation*: a projection may identify a large number of $\underset{A}{\rightarrow}$-reductions in A with a single $\underset{B}{\rightarrow}$-reduction in B.

Our notion of projection is closely related to, but weaker, than the notion of a *Galois connection*: if

- $\rhd\!\!-$ is a projection function of $\underset{A}{\rightarrow}$ into $\underset{B}{\rightarrow}$, and
- a function $\rightarrow$ from B to A is given such that $\leftarrow\ \subseteq \rhd\!\!-$,

then the pair of $\rhd\!\!-$ and $\rightarrow$ is a Galois connection between $\underset{A}{\rightarrow}$ and $\underset{B}{\rightarrow}$. The essential difference is thus that a Galois connection makes some elements of A special: all elements $a \in A$ are related to one of these $a' \in A$ throgh $a \rhd\!\!- \cdot \rightarrow a'$.

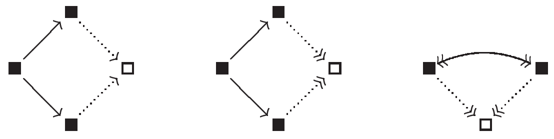
The special case of this when $A \supseteq B$ (such that the translation $\rhd\!\!-$ is itself a reduction on A) is particularly simple and has a standard name for transitive closures.

A.1.12 Definition (conservative extension). Given reductions $\underset{A}{\rightarrow}$ and $\underset{B}{\rightarrow}$ on A and B, respectively, and a projection $\rhd\!\!-$ from $\underset{A}{\twoheadrightarrow}$ to $\underset{B}{\twoheadrightarrow}$. If $A \supseteq B$ then $\underset{A}{\rightarrow}$ is a *conservative extension* of $\underset{B}{\rightarrow}$, and $\underset{B}{\rightarrow}$ is a *subreduction* of $\underset{A}{\rightarrow}$. The notions generalise in the obvious way to arbitrary relations.

Notice that translation and projection are properties of the actual reduction step, hence require a one-to-one correspondence between the length of the reductions whereas the notions of conservative extension 'only' require this of the transitive, reflexive closures.

Finally, we introduce the concepts that will be the main object of study for relations in this tutorial: confluence and normalisation.

A.1.13 Definition (confluence).



$\to\!\Diamond$ *diamond property* | $\to$LC *local confluence* | $\to$CR *Church-Rosser*

We say that $\to$ is *confluent* when $\twoheadrightarrow\!\Diamond$. In Klop (1992), LC is denoted WCR (for 'weak CR').

A classical result is the following.

A.1.14 Proposition. $\to$ is confluent if and only if $\to$CR.

Proof. The 'if' direction is trivial. The 'only if' direction is proved by assuming $\twoheadrightarrow\!\Diamond$ and consider a conversion $a \twoheadleftarrow\!\!\twoheadrightarrow b$: this must have the shape

$$a \twoheadrightarrow a_{(,1)} \twoheadleftarrow a_1 \twoheadrightarrow a_{(1,2)} \twoheadleftarrow a_2 \cdots a_n \twoheadrightarrow a_{(n,)} \twoheadleftarrow b$$

for some n. By $\twoheadrightarrow\!\Diamond$ we then have

$$a \twoheadrightarrow a_{(,1)} \twoheadrightarrow a_{((,1),(1,2)} \twoheadleftarrow a_{(1,2)} \twoheadrightarrow a_{((1,2),(2,3)} \twoheadleftarrow a_{(2,3)} \cdots \twoheadleftarrow b$$

and by induction there is a common reduct and hence $\to$CR. This kind of proof can also be done using 'diagram induction' which we will return to in the next section (first proof by Church and Rosser 1935). $\square$

A.1.15 Definitions (normalisation properties). Let $\to$ be a relation from A to B.

1. The *normal forms* $\to$-nf are those $x \in A$ for which $\nexists y : x \to y$.
2. The *final restriction* $\to\!|$ is defined by $w \to\!| \, x$ iff $w \to x$ and x is an $\to$-nf.
3. $\to$ has *unique normal forms*, written $\to$UN, if $\twoheadrightarrow\!|$ is a function.
4. The set of $\to$-*weakly normalising* objects is the subset $\mathrm{WN}_{\to} \subseteq A$ defined by $\mathrm{WN}_{\to} = \{ x \in A \mid \exists y : x \twoheadrightarrow\!| \, y \}$. We say '$\to$ *is weakly normalising*', and write $\to$WN, when $\mathrm{WN}_{\to} = A$.
5. $\rightsquigarrow$ is said to *preserve weak normalisation* (PWN) of $\to$ if $\mathrm{WN}_{\rightsquigarrow} \supseteq \mathrm{WN}_{\to}$.

6. The set of $\rightarrow$-*strongly normalising* objects is the subset $\mathrm{SN}_{\rightarrow} \subseteq A$ defined such that $x \in \mathrm{SN}_{\rightarrow}$ implies that all reductions $x \rightarrow x_1 \rightarrow \cdots$ from x are finite. We say '$\rightarrow$ *is strongly normalising*', and write $\rightarrow$SN, when $\mathrm{SN}_{\rightarrow} = A$.

7. $\rightsquigarrow$ is said to *preserve strong normalisation* (PSN) of $\rightarrow$ if $\mathrm{SN}_{\rightsquigarrow} \supseteq \mathrm{SN}_{\rightarrow}$.

8. $\rightarrow$ is *complete* if it is CR and SN.

A.1.16 History. Huet (1980) uses *noetherian* (after Emmy Noether) for SN; sometimes *terminating* is used. Some use $\rightarrow^!$ for what we write as $\twoheadrightarrow\!\!|$. Klop (1992) uses $\mathrm{UN}^{\rightarrow}$ for what we call UN.

The reasons why PSN and PWN are interesting in their own right, and the reason there is no 'preservation of $\diamondsuit$', is the following.

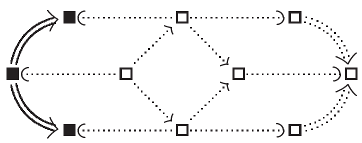
A.1.17 Propositions. Given reductions $\rightarrow$ and $\Rrightarrow$ with a translation $\multimap$ from $\rightarrow$ to $\Rrightarrow$.

1. $\rightarrow\diamondsuit$ iff $\Rrightarrow\diamondsuit$

2. If $\multimap$ is a projection then $\Rrightarrow$WN implies $\rightarrow$WN

3. If $\multimap$ is a projection then $\rightarrow$SN implies $\Rrightarrow$SN

Proofs. 1 is proved by an instance of what is called the 'interpretation method' by Hardin; we show the 'only if' case (the 'if' is symmetric).



2 follows from the fact that we *remove* reduction elements when taking a subreduction so no infinite ones can be added. 3 by the converse argument: since a conservative extension preserves reductions then it also preserves the normalising ones. $\square$

A.2 Inductive Notations

This section defines the basic inductive notations and terminology used in this tutorial. Such a fundamental common lingo for inductive reasoning makes many arguments in the sequel shorter and their presentation more uniform.

The presentation follows computer science tradition.

First we define sequences that are suitable for *length induction*; in case of finite sequences this is even *well-founded*; we include some properties of sequences including how to interpret them as maps. Then we define inductively defined sets suitable for *structural induction*, including the associated notions of *contextuality* for induction over terms and the weaker *compositionality* for induction over derivations. Finally, we generalise the diagram reasoning to also include induction.

A.2.1 Notations (sequences).

1. A *finite sequence* over Φ, $\phi = (\phi_1, \ldots, \phi_n)$, with elements $\phi_i \in \Phi$, has *length* $\#\phi = n \in \mathbb{N}_0$. The empty sequence is $\epsilon = ()$ (with $\#\epsilon = 0$), and the following derived sequences are defined

$$(\phi_1, \ldots, \phi_n) \cdot (\psi_1, \ldots, \psi_m) = (\phi_1, \ldots, \phi_n, \psi_1, \ldots, \psi_m) \qquad \text{(concatenate)}$$

$$f \circ (\psi_1, \ldots, \psi_n) = (f(\psi_1), \ldots, f(\psi_n)) \qquad \text{(map)}$$

$$(\phi_1, \ldots, \phi_n)\backslash m = \begin{cases} \epsilon & \text{for } m = 0 \\ (\phi_1, \ldots, \phi_m) & \text{for } 0 < m \le n \\ (\phi_1, \ldots, \phi_n) & \text{for } n < m \end{cases} \qquad \text{(take)}$$

$$(\phi_1, \ldots, \phi_n)/m = \begin{cases} (\phi_1, \ldots, \phi_n) & \text{for } m = 0 \\ (\phi_{m+1}, \ldots, \phi_n) & \text{for } 0 < m \le n \\ \epsilon & \text{for } n < m \end{cases} \qquad \text{(drop)}$$

where f in (map) must be a function whose domain includes the elements of ψ, and $m \in \mathbb{N}_0$ (so we can take or drop 0 elements from a sequence). The set of finite sequences over Φ is denoted Φ^*.

2. An *infinite sequence* is written $\phi = (\phi_1, \phi_2, \ldots)$ and has $\#\phi = \infty$ and $\phi(i) = \phi_i$ for $1 \le i$. The derived sequences are as in the finite case except that

$$(\phi_1, \ldots) \cdot (\psi_1, \ldots, \psi_m) = (\phi_1, \ldots) \qquad \text{(concatenate-}\infty\text{)}$$

The *identity sequence* is the infinite sequence $\iota = (1, 2, 3, \dots)$ enumerating all elements of $\mathbb{N}$.

3. A sequence over Φ can be used as a *function* (or *map*) from $\mathbb{N}$ to Φ through the convention that $\phi(i) = \phi_i$ for $1 \le i \le n$. ϵ corresponds to the everywhere undefined *empty map*; the identity sequence $\iota = (1, 2, 3, \dots)$ similarly corresponds to the *identity map* on $\mathbb{N}$. With this interpretation, f in the (map) rule can also be a sequence.

We write $x \in \phi$ when $\exists i : \phi(i) = x$, and a sequence ϕ is *natural* if it is over $\mathbb{N}$.

The sequence operations satisfy several simple algebraic properties that we will use freely; they also suggest which parentheses we can leave out without risk of ambiguity. We will also confuse x and the singleton sequence (x) when it is apparent what is intended, we will occasionally use a different element separator than ',' (comma), and we will sometimes denote sequences as vectors $\overrightarrow{\phi}$.

A.2.2 Propositions (sequence algebra).

1. $\phi \cdot (\psi \cdot \gamma) = (\phi \cdot \psi) \cdot \gamma$.

2. $\epsilon \cdot \phi = \phi \cdot \epsilon = \phi$.

3. $\omega \cdot \psi = \omega$ for ω infinite.

4. $\epsilon \circ \phi = \phi \circ \epsilon = \epsilon$.

5. $\phi \circ \iota = \phi$; if ϕ is natural then $\iota \circ \phi = \phi$.

6. $\phi \circ (\psi \cdot \gamma) = (\phi \circ \psi) \cdot (\phi \circ \gamma)$.

7. If $m + n \le \#\phi$ then $\phi/m\backslash n = \phi\backslash(m+n)/m$.

8. $(\iota/n)(m) = n + m$ for $n, m > 0$.

9. $(\phi \cdot \psi)\backslash m = \begin{cases} \phi\backslash m & \text{if } m < \#\phi \\ \phi \cdot (\psi\backslash(m - \#\phi)) & \text{otherwise} \end{cases}$.

10. $(\phi \cdot \psi)/m = \begin{cases} \phi/m \cdot \psi & \text{if } m < \#\phi \\ \psi/(m - \#\phi) & \text{otherwise} \end{cases}$.

11. $\phi\backslash m = \phi \circ (1, \dots, m)$.

12. $\phi/m = \phi \circ (m + 1, \dots)$.

Next we establish a concise way of defining sets of *terms* where we can use *structural induction*. In practice we will usually distinguish between a purely inductive (context-free) definition of *preterms* followed by a grouping of these into terms by some equivalence; a preterm is said to be *representative* of the corresponding term.

A.2.3 Notation (inductive structures). When a set is 'inductively defined by ... ' we use the following metanotation: $\mathsf{X} ::= \dots$ is a *production* that reads 'X may be constructed as ...'. Several alternative productions can be given for the same X, allowing X to be constructed in more than one way; for convenience $\mathsf{X} ::= \mathsf{Y} \mid \mathsf{Z}$ is a shorthand for the two productions $\mathsf{X} ::= \mathsf{Y}$ and $\mathsf{X} ::= \mathsf{Z}$.

The *size* of a term t, written $|\mathsf{t}|$, is the *number of symbols* in t.

As a special convenience we allow the use of *Kleene's star* in productions: X^* denotes arbitrary *sequences* (in the above sense) of terms of the form X, and we use ϵ instead of nothing to denote 'empty'.

A.2.4 History. The inductive notation, including the symbols '$::=$' and '$\mid$', originates in *Backus-Naur form* (BNF), the 'metalinguistic formulae' used to specify the ALGOL programming language (Naur et al. 1960). However, since we are only concerned with *abstract syntax*, we do not use the original '$\langle\cdots\rangle$' markers for variables over terms but rely on mathematical conventions. Such a definition is also known as a *context-free grammar*. Kleene's star was originally a notation for regular expressions; in so-called *extended BNF* this is usually denoted by braces as $\{\mathsf{X}\}$.

For each notion of terms it is natural to consider arbitrary subterms and contexts.

A.2.5 Definitions (contextuality). Given a notion of T-term defined inductively by some $\mathsf{t} ::= \dots$ productions.

1. A T-*context* is defined by reusing the same definition but augmenting the production to $\mathsf{t} ::= \cdots \mid \Box$, where $\Box$ is a special, unique symbol called *hole*, and considering only terms with exactly one occurrence of $\Box$. $\mathsf{C}[\mathsf{t}]$ is the T-term obtained by inserting t in place of the hole.

2. If for some T-terms t, t' there is a T-context $\mathsf{C}[\,]$ such that $\mathsf{t} = \mathsf{C}[\mathsf{t}']$ then t' is a *subterm* of t, and conversely, t is a *superterm* of t'. The induced *subterm ordering* is written $\mathsf{t}' \trianglerighteq \mathsf{t}$; it is a partial order.[4]

[4]The intuition is that it 'points to the root' with the 'subterm at the bottom'.

3. The *strict subterm ordering* is written $t' \rhd t$ and defined by $t' \trianglerighteq t \wedge t' \neq t$.

4. A predicate $p(t)$ is *contextual* if $p(t)$ implies $p(C[t])$ for any context $C[]$.

This generalises to any finite number of distinctly labeled holes: using $\boxed{1}, \ldots, \boxed{n}$ we let $C[\vec{t}]$ denote insertion of the terms of the n-ary vector of terms $\vec{t}$ in place of the corresponding holes.

An example of this notion is *contextual reduction* (*compatible relation* in Huet 1980): $\to$ on an inductively defined set T is contextual if $t \to t'$ implies $C[t] \to C[t']$ for any T-context $C[]$.

When dealing with non-contextual properties we can not in general use induction over the term structure. However, most properties we will define will still be 'semi-contextual' in the sense that they are defined inductively by cases over the term constructors in a 'logical' style by inference rules. This is captured by the following:

A.2.6 Definitions (compositionality). Suppose a notion of T-term is defined inductively by the production

$$t ::= D_1[\vec{t}_1] \mid \cdots \mid D_n[\vec{t}_n] \qquad (*)$$

where $D_i[]$ are T-contexts. Consider a predicate p over n-tuples of T-terms.

1. p is *flat* if it is defined by a collection of *axioms* of the form

$$p(C[\vec{x}])$$

   for T-contexts $C[]$.

2. p is *compositional* if it is the disjunction (union) of a flat predicate p_0 and a set of *inference rules* of the form

$$\frac{p_1(C_1[\vec{x}], \vec{y}) \ \wedge \ \ldots \ \wedge \ p_n((C_n[\vec{x}], \vec{y})}{p(D_i[\vec{x}], \vec{y})}$$

   where the $C_j[\vec{x}] \rhd D_i[\vec{x}]$, $i \in \{1, \ldots, n\}$; $\vec{x}$ and $\vec{y}$ are the *inference metavariables* where those in $\vec{x}$ must be pairwise distinct, and $C[]$ is any T-context.

3. p is *semicompositional* if the same holds except we relax the strict subterm relation $\rhd$ to $\trianglerighteq$, allowing the entire term of the conclusion to appear in the premise.

These generalise the natural way to any finite number of mutually recursive productions.

In this tutorial virtually all 'predicates' we study will be binary relations where we write '$x \rightarrow x'$' instead of $p(x, x')$; the corresponding notions of *compositional reduction* and *flat reduction* are common as is the notion of *derivor* that means a *compositional function*.

A.2.7 Remark. Compositionality as defined here is very restrictive, and rules out many inference systems that are normally accepted. In particular, implicit fixed point iterations as used, *e.g.*, we rule out the recursion rules of the 'natural semantics' of Kahn (1987), and non-local operations such as the $\hat{\ }$ 'renaming transformation' of Launchbury (1993).

Note, however, that any contextual property p is also compositional because rules such as

$$\frac{p(x_j)}{p(D_i[x_1, \ldots, x_n])}$$

can be derived from the contextuality.

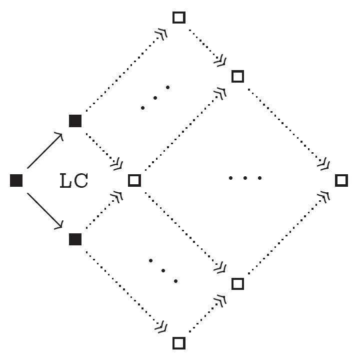
Finally, we present an induction principle for diagrams.

A.2.8 Notation (diagrammatic induction). An inductive sequence of diagrams can be compacted into a single picture by adding dots from a solid relation indicating that the step leading to it should be repeated by duplicating the 'step cell' in the indicated direction (one step per diagram in the compacted sequence).

This is of course only meaningful when the diagram 'cell repetition' is guaranteed to be well defined. The most used confluence result uses SN to ensure that double induction is possible:

A.2.9 Lemma (Newman's lemma). If $\rightarrow$SN and $\rightarrow$LC then $\rightarrow$CR (Newman 1942).

Proof. Double induction:



$\square$

A.3 λ-Calculus

The λ-calculus is the simplest and most concise model of the computable functions. It was also the main inspiration for functional programming languages, and can thus be seen as a (very) minimal functional language – at least many of the interesting problems of functional programming languages can be studied within the λ-calculus.

The λ-calculus was invented by Church (1941); this section summarises the syntax of the λβ-calculus following Barendregt (1984) except we use the inductive notations summarised in section A.2.

A.3.1 Definition (λ-preterms). The set of λ-preterms is ranged over by the letters $MNPQ$ and R, and defined inductively by

$$M ::= x \mid \lambda x.M \mid MN$$

where the letters $xyzv$ and w range over an infinite set of variables. A preterm of form $\lambda x.M$ is called an *abstraction* and a preterm of form MN is called an *application*. We use parentheses to indicate subpreterm structure and write $\lambda xy.M$ for $\lambda x.(\lambda y.M)$ and MNP for $(MN)P$ with the latter having higher precedence, *i.e.*, $\lambda xy.MNP$ abbreviates $\lambda x.(\lambda y.((MN)P))$.

Several auxiliary concepts are needed to change the view from the context-free preterms to the usual notion of λ-terms which includes variable binding.

A.3.2 Definitions (λ-terms).

1. The *free variable* set of a λ-term M is denoted $\mathrm{fv}(M)$ and defined inductively over M by $\mathrm{fv}(x) = \{x\}$, $\mathrm{fv}(\lambda x.M) = \mathrm{fv}(M)\setminus\{x\}$, and $\mathrm{fv}(MN) = \mathrm{fv}(M) \cup \mathrm{fv}(N)$. A λ-preterm M is *closed* iff $\mathrm{fv}(M) = \varnothing$.[5]

2. The result of *renaming* all free occurrences of y in M to z is written $M[y := z]$ and defined inductively over M by $x[y := z] = z$ if $x = y$, $x[y := z] = x$ if $x \neq y$, $(\lambda x.M)[y := z] = \lambda x'.M[x := x'][y := z]$ with $x' \notin \mathrm{fv}(\lambda x.M) \cup \{y, z\}$, and $(MN)[y := z] = (M[y := z])\,(N[y := z])$.

3. Two preterms are *α-equivalent*, written $M \equiv N$, when they are identical except for *renaming* of bound variables, defined inductively by $x \equiv x$, $\lambda x.M \equiv \lambda y.N$ if $M[x := z] \equiv N[y := z]$ for $z \notin \mathrm{fv}(MN)$, and $MN \equiv PQ$ if $M \equiv P \wedge N \equiv Q$.

[5] We even allow $\mathrm{fv}(C[\,])$ for any context $C[\,]$ by the convention that $\mathrm{fv}(\square) = \varnothing$.

4. The set of λ-*terms*, Λ, is the set of λ-preterms modulo $\equiv$. The set of *closed* λ-*terms*, Λ°, is the subset of Λ where the representatives are closed preterms.

The definition of Λ° only makes sense because of the following easily proved property:

A.3.3 Proposition. For preterms M and N, $M \equiv N$ implies $\mathrm{fv}(M) = \mathrm{fv}(N)$.

This means that we can safely generalise the notion of free variables and renaming to λ-terms – in fact it means that we can generally confuse preterms and terms which is fortunate because we wish to observe the behaviour of λ-terms but can only write down (concrete) preterms. To ease the presentation we will follow the usual convention of Barendregt (1984, 2.1.13) which resolves naming conflicts by always picking an appropriate representative preterm for a term.

A.3.4 Convention (bound variable naming). When a group of preterms 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.

A.3.5 Remark. The convention we use is slightly stronger than Barendregt's in that explicitly distinct bound variables are required to be truly distinct. This is a convenience to avoid not only capture but also variable clash as we shall see below.

Now we are ready to define $\lambda\beta$-reduction (we will make free use of the standard reduction relation notation summarised in A.1).

A.3.6 Definition (β-reduction). The reduction relation $\underset{\beta}{\rightarrow}$ is the contextual closure of the relation generated by the rule

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

where $M[x := N]$ is the term obtained by substituting N for all free occurrences of x in M, defined inductively by

$$\begin{aligned} x[y := N] &\equiv \begin{cases} N & \text{if } x = y \\ x & \text{if } x \neq y \end{cases} \\ (\lambda x.M)[y := N] &\equiv \lambda x.M[y := N] \qquad\qquad (*) \\ (M_1 M_2)[y := N] &\equiv M_1[y := N]\ M_2[y := N] \end{aligned}$$

The variable convention is crucial to avoid two problems in $(*)$ above:

Variable clash. The requirement that bound variables are distinct means that $x \neq y$ will always be true when building $(\lambda x.M)[y := N]$, *e.g.*, to prevent breaking the distinction $(\lambda x.x)[x := z] \not\equiv \lambda x.x[x := z]$.

Variable capture. The requirement that bound and free variables are distinct means that $x \notin \mathrm{fv}(N)$ and thus prevents capturing free variables, *e.g.*, to prevent breaking the distinction $(\lambda x.y)[y := x] \not\equiv \lambda x.y[y := x]$.

A.3.7 Example. To give the reader an idea of the reduction relation, the full λβ-reduction graph of the term $(\lambda y.(\lambda z.z)y)x$ is $(\lambda y.(\lambda z.z)y)x \overset{\beta}{\underset{\beta}{\rightrightarrows}} (\lambda z.z)x \xrightarrow[\beta]{} x$. Notice how the two reductions from the first would result in different namings that denote the same term (the naming shown is the one obtained from the top reduction).

A.3.8 Notation (standard combinators). A closed λ-term is called a *combinator*, and a growing number of such have been given 'standard' names:

$$
\begin{gathered}
I \equiv \lambda x.x \;\;;\;\; K \equiv \lambda xy.x \;\;;\;\; K^* \equiv \lambda xy.y \;\;;\;\; S \equiv \lambda xyz.xz(yz) \\
\Omega \equiv \omega\omega \;\;;\;\; \omega \equiv \lambda x.xx \quad (\omega \text{ also called `self'}) \\
Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \quad (Y \text{ is the `Curry fixed point combinator'}) \\
\Theta \equiv AA \;;\; A \equiv \lambda xy.y(xxy) \quad (\Theta \text{ is the `Turing fixed point combinator'})
\end{gathered}
$$

A.3.9 Remark (extensionality). The origin of λ-calculus is as a vehicle for expressing the recursive functions, and this is the interest we have in the λ-calculus in this tutorial. However, when studying the relation to *logic* the following additional rewrite rule is important:

$$\lambda x.Mx \to M \quad \text{if } x \notin \mathrm{fv}(M) \tag{η}$$

We will not discuss (η) much in this tutorial.

Finally we reproduce some standard theorems about λβ-calculus:

A.3.10 Proposition (substitution lemma).

$$M[x := N][y := P] \equiv M[y := P][x := N[y := P]]$$

Proof. See Barendregt (1984, 2.1.16). $\square$

A.3.11 Theorem (Church-Rosser). $\underset{\beta}{\rightarrow}$ CR.

Proof. See Church and Rosser (1935, Theorem 1). $\square$

Bibliography

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

Seminal paper introducing explicit substitution, discussing theory, including a type system, and applications, including the derivation of an abstract machine.

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

Semantic study of the λ-calculus restricted to weak leftmost outermost reduction (somewhat confusingly called 'lazy').

Aczel, P. (1978). A general Church-Rosser theorem. *Technical report*. Univ. of Manchester.

Early treatment of higher-order rewriting.

American Mathematical Society (1995). *AMS-LaTeX Version 1.2 User's Guide*. ⟨URL: ftp://ftp.tex.ac.uk/ctan/tex-archive/macros/ams/amslatex/⟩

Ariola, Z. M. and Arvind (1992). Graph rewriting systems. *Computation Structures Group Memo 323-1*. MIT.

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.

A study of how to model sharing through binding using let.

Ariola, Z. M. and Klop, J. W. (1994). Cyclic lambda graph rewriting. *LICS '94—Proc. Ninth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press. Paris, France. pp. 416–425. full version forthcoming (Ariola and Klop 1995b).

A study of λ-graph rewriting using a recursive binding construct to represent sharing and cycles.

Ariola, Z. M. and Klop, J. W. (1995a). Equational term graph rewriting. *Technical report*. CWI. To appear in Acta Informatica.

A study of how recursive equations can be implemented with term graph rewriting using proper cycles to represent recursion in the equations.

Ariola, Z. M. and Klop, J. W. (1995b). Lambda calculus with explicit recursion. Personal Communication.

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

Standard reference work on untyped λ-calculus.

Barendregt, H. P., van Eekelen, M. C. D. J., Glauert, J. R. W., Kennaway, J. R., Plasmeijer, M. J. and Sleep, M. R. (1987). Term graph rewriting. *In* de Bakker, J. W., Nijman, A. J. and Treleaven, P. C. (eds), *PARLE '87—Parallel Architectures and Languages Europe vol. II*. Number 256 in *LNCS*. Springer-Verlag. Eindhoven, The Netherlands. pp. 141–158.

Benaissa, Z.-E.-A., Briaud, D., Lescanne, P. and Rouyer-Degli, J. (1995). λυ, a calculus of explicit substitutions which preserves strong normalisation. *Rapport de Recherche 2477*. INRIA, Lorraine. Technôpole de Nancy-Brabois, Campus Scientifique, 615 rue de Jardin Botanique, BP 101, F-54600 Villers lès Nancy. ⟨URL: http://www.loria.fr/~lescanne/PUBLICATIONS/RR-2477.PS⟩

Thorough treatment of λυ, including the PSN proof.

Benaissa, Z.-E.-A. and Lescanne, P. (1996). Triad machine: A general computation model for the description of abstract machines. Personal communication.

Benaissa, Z.-E.-A., Lescanne, P. and Rose, K. H. (1996). Modeling sharing and recursion for weak reduction strategies using explicit substitution. *In* Kuchen, H. and Swierstra, D. (eds), *PLILP '96—Programming Languages: Implementation, Logics and Programs*. LNCS. Springer-Verlag. Aachen, Germany. pp. 393–407. ⟨URL: http://www.loria.fr/~kris/benaissa+lescanne+rose-96.ps.gz⟩

Shows how explicit substitution, binding, and addressing, can combine with weak reduction to create a class of low-level computational models for functional programming languages with properties desirable for reasoning about program transformations.

Bloo, R. and Geuvers, J. H. (1996). Explicit substitution: on the edge of strong normalisation. *Computing Science Reports 96–10*. Eindhoven University of Technology. P.O.box 513, 5600 MB Eindhoven, The Netherlands.

Study of the use of recursive path ordering to show PSN for explicit substitution calculi.

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*. pp. 62–72. ⟨URL: ftp://ftp.diku.dk/diku/semantics/papers/D-246.ps⟩

Gives a direct proof of PSN for λx.

Bloo, R. and Rose, K. H. (1996). Combinatory reduction systems with explicit substitution that preserve strong normalisation. *In* Ganzinger, H. (ed.), *RTA '96—Rewriting Techniques and Applications*. Number 1103 in *LNCS*. Springer-Verlag. Rutgers University, New Jersey. pp. 169–183. ⟨URL: http://www.brics.dk/~krisrose/PAPERS/bloo+rose-rta96.ps.gz⟩

Formalises the notion of explicit substitution as a syntactic constraint on CRSs. Shows how any CRS can be transformed into an explicit substitution conservative extension, and shows how this even preserves strong normalisation for a large class of CRSs (the "structure-preserving" CRSs).

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. Also chapter C.2 of (Nederpelt, Geuvers and de Vrijer 1994).

Seminal paper introducing what is now known as de Bruijn indices & levels.

Burn, G. L., Peyton Jones, S. L. and Robson, J. D. (1988). The spineless G-machine. *LFP '88—ACM Conference on LISP and Functional Programming*. Snowbird, Utah. pp. 244–258.

Church, A. (1941). *The Calculi of Lambda-Conversion*. Princeton University Press. Princeton, N. J.

The book on the λ-calculus from the source.

Church, A. and Rosser, J. B. (1935). Some properties of conversion. *Transactions on the AMS* 39: 472–482.

The original Church-Rosser proof.

Crégut, P. (1990). An abstract machine for the normalisation of λ-terms. *LFP '90—ACM Conference on LISP and Functional Programming*. Nice, France. pp. 333–340.

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

Uses categorical concepts to obtain a family of finite, first-order models of λ-calculi with particular reduction strategies.

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

Study of the relation between de Bruijn indices and the use of an environment.

Curien, P.-L., Hardin, T. and Lévy, J.-J. (1996). Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM* 43(2): 362–397.

Study of the extensions of $\lambda\sigma$ needed to obtain a calculus which is confluent on 'open' terms (*i.e.*, on terms permitting metavariables for terms and substitutions.

Curry, H. B. and Feys, R. (1958). *Combinatory Logic*. Vol. I. North-Holland.

Seminal book on logic and λ-calculus. Defines the programme that lead to explicit substitution.

Danvy, O. and Pfenning, F. (1995). The occurrence of continuation parameters in cps terms. *Technical Report CMU-CS-95-121*. School of Computer Science, Carnegie Mellon University. Pittsburgh, PA 15213.

Dershowitz, N. (1979). A note on simplification orderings. *Information Processing Letters* 9(5): 212–215.

Invents the powerful "simplification ordering" technique (now known as "recursive path ordering") for proving termination.

Felleisen, M. and Friedman, D. P. (1989). A syntactic theory of sequential state. *Theoretical Computer Science* 69: 243–287.

First use of global address annotations on λ-like terms.

Glauert, J. R. W., Kennaway, J. R. and Sleep, M. R. (1989). Final specification of Dactl. *Report SYS–C88–11*. University of East Anglia. Norwich, U.K.

Gordon, A. D. (1993). A mechanisation of name-carrying syntax up to alpha-conversion. *HUG'93—Proc. of International Higher-Order Logic theorem Proving Workshop*. LNCS. Univ. of British Columbia. ⟨URL: ftp://ftp.cl.cam.ac.uk/papers/adg/hug93.dvi.gz⟩

Grue, K. (1987). Call-by-mix: A reduction strategy for pure λ-calculus. Unpublished note from DIKU (University of Copenhagen).

Hardin, T., Maranget, L. and Pagano, B. (1996). Functional back-ends within the lambda-sigma calculus. *FPCA '96—Functional Programming Languages and Computer Architecture*. ACM Press. Philadelphia, Pennsylvania, USA. pp. 25–33.

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

Important book with first thorough treatment of "lazy" evaluation.

Huet, G. (1980). Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM* 27(4): 797–821.

Terminological paper fixing many concepts in rewriting.

Hughes, J. M. (1982). Super-combinators: A new implementation method for applicative languages. *LFP '82—ACM Symposium on LISP and Functional Programming*. Pittsburgh, Pensylvania. pp. 1–10.

Jeffrey, A. (1993). A fully abstract semantics for concurrent graph reduction. *Computer Science Report 12/93*. School of Cognitive and Computing Sciences, University of Sussex. Falmer, Brighton BN1 9QH, UK.

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

Kahn, G. (1987). Natural semantics. *Rapport 601*. INRIA. Sophia-Antipolis, France.

Kamareddine, F. and Nederpelt, R. P. (1993). On stepwise explicit substitution. *International Journal of Foundations of Computer Science* 4(3): 197–240.

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

Presents λs with CR and PSN results.

Kamareddine, F. and Ríos, A. (1996). Bridging de Bruijn indices and variable names in explicit substitution calculi. Personal communication.

Analysis of several explicit substitution calculi through comparison to λx.

Kennaway, J. R., Klop, J. W., Sleep, M. R. and de Vries, F. J. (1995). Transfinite reductions in orthogonal term rewriting systems. *Information and Computation* 119(1): 18–38.

Kennaway, J. R. and Sleep, M. R. (1988). Director strings as combinators. *ACM Transactions on Programming Languages and Systems* 10: 602–626.

Klop, J. W. (1980). *Combinatory Reduction Systems*. PhD thesis. University of Utrecht. Also available as Mathematical Centre Tracts 127.

Seminal introduction of CRSs and orthogonality.

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

Standard reference on rewriting.

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

CRSs ten years after (Klop 1980): a few new proofs and directions for further study that are all very active now.

Knuth, D. E. (1973). *Fundamental Algorithms*. Vol. 1 of *The Art of Computer Programming*. second edn. Addison-Wesley.

Standard work on algorithms.

Knuth, D. E. (1984). *The TEXbook*. Addison-Wesley.

Koopman, P. W. M., Smetsers, S., van Eekelen, M. C. D. J. and Plasmeijer, M. J. (1991). Efficient graph rewriting using the annotated functional strategy. *In* Plasmeijer and Sleep (1991). pp. 225–250. (available as Nijmegen Tech. Report 91-25).

Lamport, L. (1994). *LATEX—A Document Preparation System*. 2nd edn. Addison-Wesley.

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

Seminal work with the first abstract machines: SECD.

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.

Early attempt of using abstract addresses properly for lazy evaluation.

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

Survey of explicit substitution calculi all the way back to unpublished work of de Bruijn.

Lescanne, P. and Rouyer-Degli, J. (1995). Explicit substitutions with de Bruijn's levels. *In* Lescanne, P. (ed.), *RTA '95—Rewriting Techniques and Applications*. Number 914 in *LNCS*. Springer-Verlag. Kaiserslautern, Germany. pp. 294–308.

Presents $\lambda\chi$ with CR and PSN results.

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

Seminal thesis defining many important properties, notably *optimal* and *parallel* reduction.

McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P. and Levin, M. (1965). *Lisp 1.5 Programmer's Manual*. MIT Press. Cambridge, Mass.

First funcitonal programming language!

Melliès, P.-A. (1995). Typed λ-calculi with explicit substitution may not terminate. *In* Dezani, M. (ed.), *TLCA '95—Int. Conf. on Typed Lambda Calculus and Applications*. Vol. 902 of *LNCS*. Springer-Verlag. Edinburgh, Scotland. pp. 328–334.

Milner, R., Tofte, M. and Harper, R. (1990). *The Definition of Standard ML*. MIT Press.

Muñoz Hurtado, C. A. (1996). Confluence and preservation of strong normalisation in an explicit substitutions calculus. *LICS '96—Proc. 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press. Rutgers, New Jersey.

Describes a particular reduction strategy that gives PSN and confluence on open terms while still allowing for some nodeterminism.

Naur, P. and others (1960). Report on the algorithmic language ALGOL 60. *Communications of the ACM* 3: 299–314.

Nederpelt, R. P., Geuvers, J. H. and de Vrijer, R. C. (eds) (1994). *Selected Papers on Automath*. Vol. 133 of *Studies in Logic*. North-Holland.

Newman, M. H. A. (1942). On theories with a combinatorial definition of "equivalence". *Annals of Mathematics* 43(2).

First (complicated) proof of Newman's Lemma A.2.9.

Nipkow, T. (1991). Higher-order critical pairs. *LICS '91—Proc. Sixth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press. Amsterdam, The Netherlands. pp. 342–349.

van Oostrom, V. (1994). *Confluence for Abstract and Higher-Order Rewriting*. PhD thesis. Vrije Universiteit, Amsterdam.

van Oostrom, V. and van Raamsdonk, F. (1995). Weak orthogonality implies confluence: the higher-order case. *Technical Report CS-R9501*. CWI.

Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages*. Prentice-Hall.

Peyton Jones, S. L. (1992). Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming* 2(2): 127–202.

Thorough treatment of how one can implement lazy functional languages on real computers.

Plasmeijer, M. J. and Sleep, M. R. (eds) (1991). *SemaGraph '91—Symposium on the Semantics and Pragmatics of Generalized Graph Rewriting*. Katholieke Universiteit Nijmegen. Nijmegen, Holland. (available as Nijmegen Tech. Report 91-25).

Plotkin, G. D. (1975). Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science* 1: 125–159.

Seminal paper on strategies and their realisation by CPS.

Plotkin, G. D. (1977). LCF considered as a programming language. *Theoretical Computer Science* 5: 223–255.

Seminal paper on operational semantics.

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

Important description of the practical considerations one should be aware of when discussing structural operational semantics.

Purushothaman, S. and Seaman, J. (1992). An adequate operational semantics of sharing in lazy evaluation. *In* Krieg-Brückner, B. (ed.), *ESOP '92—4th European Symposium on Programming*. Number 582 in *LNCS*. Springer-Verlag. Rennes, France. pp. 435–450.

Revesz, G. (1985). Axioms for the theory of lambda-conversion. *SIAM Journal on Computing* 14(2): 373–382.

Rose, E. (1996). Linear time hierarchies for a functional language machine model. *In* Nielson, H. R. (ed.), *ESOP '96—6th European Symposium on Programming*. Number 1058 in *LNCS*. Springer-Verlag. Linköping, Sweden. pp. 311–325. ⟨URL: ftp://ftp.diku.dk/diku/semantics/papers/D-270.ps⟩

Thorough study of the relationship betwen fine-grained complexity measures for first-order and higher-order functional programming.

Rose, K. H. (1992). Explicit cyclic substitutions. *In* Rusinowitch, M. and Rémy, J.-L. (eds), *CTRS '92—3rd International Workshop on Conditional Term Rewriting Systems*. Number 656 in *LNCS*. Springer-Verlag. Pont-a-Mousson, France. pp. 36–50. ⟨URL: ftp://ftp.diku.dk/diku/semantics/papers/D-143.ps⟩

Presents conditional term rewrite system models of the λ-calculus enriched with first non-recursive and then recursive local definitions realised by explicit cyclic subsitution.

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

Combines structural operational semantics and term graph rewriting into *Graph Operational Semantics* (GOS).

Rose, K. H. (1996a). On explicit binding and substitution preserving strong normalisation. Submitted. ⟨URL: http://www.brics.dk/~krisrose/PAPERS/rose-on-explicit-binding.ps.gz⟩

Shows how the representation of explicit binding in is not essential for general properties of reduction for explicit substitution calculi.

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

Study of the operational semantics of functional programming languages, specifically sharing and recursion, based on explicit substitution.

Rose, K. H. and Moore, R. R. (1995). XY-pic, version 3. computer software kit. Includes User's Guide and Reference Manual. ⟨URL: ftp://ftp.tex.ac.uk/tex-archive/macros/generic/diagrams/xypic⟩

Rosen, B. K. (1973). Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM* 20(1): 160–187.

Important treatment of how to use diagrams for rewriting proofs.

Sestoft, P. (1994). Deriving a lazy abstract machine. *Technical Report ID-TR 1994-146*. Dept. of Computer Science, Technical University of Denmark. ⟨URL: ftp://ftp.dina.kvl.dk/pub/Staff/Peter.Sestoft/papers/amlazy4.dvi.gz⟩

Spiegel, M. R. (1968). *Mathematical Handbook of Formulas and Tables*. Schaum's Outline Series. McGraw-Hill.

Staples, J. (1978). A graph-like lambda calculus for which leftmost outermost reduction is optimal. *In* Claus, V., Ehrig, H. and Rozenberg, G. (eds), *1978 International Workshop in Graph Grammars and their Application to Computer Science and Biology*. Number 73 in *LNCS*. Springer-Verlag. Bad Honnef, F. R. Germany. pp. 440–454.

Staples, J. (1980). Computation on graph-like expressions. *Theoretical Computer Science* 10: 171–185.

Steele, Jr., G. L. (1978). Rabbit: A compiler for scheme. *Technical Report 474*. MIT, AI Labaratory.

First CPS based compiler.

Toyama, Y., Smetsers, S., van Eekelen, M. C. D. J. and Plasmeijer, M. J. (1991). The functional strategy and transitive term rewriting systems. *In* Plasmeijer and Sleep (1991). pp. 99–114. (available as Nijmegen Tech. Report 91-25).

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

Important publication of how to implement proper lazy languages.

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

Seminal treatment of λ-graph rewriting.

Index

This index collects definitions, concepts, people, and symbols discussed in the tutorial. The ordering is strictly alphabetical with symbols ordered according to their English spelling, *e.g.*, β and $\xrightarrow{\beta}$ are ordered as 'beta'.

M

N

O

P

R

S

Recent Publications in the BRICS Lecture Series

LS-96-3 **Kristoffer H. Rose.** ***Explicit Substitution – Tutorial & Survey*****. September 1996. v+150 pp.**

LS-96-2 **Susanne Albers.** ***Competitive Online Algorithms*****. September 1996. iix+57 pp.**

LS-96-1 **Lars Arge.** ***External-Memory Algorithms with Applications in Geographic Information Systems*****. September 1996. iix+52 pp.**

LS-95-5 **Devdatt P. Dubhashi.** ***Complexity of Logical Theories*****. September 1995. x+46 pp.**

LS-95-4 **Dany Breslauer and Devdatt P. Dubhashi.** ***Combinatorics for Computer Scientists*****. August 1995. viii+184 pp.**

LS-95-3 **Michael I. Schwartzbach.** ***Polymorphic Type Inference*****. June 1995. viii+24 pp.**

LS-95-2 **Sven Skyum.** ***Introduction to Parallel Algorithms*****. June 1995. viii+17 pp. Second Edition.**

LS-95-1 **Jaap van Oosten.** ***Basic Category Theory*****. January 1995. vi+75 pp.**