

Operational Reduction Models  
for  
Functional Programming Languages

Kristoffer Høgsbro Rose\*

*Ph.D. Thesis*

DIKU, University of Copenhagen  
Universitetsparken 1  
2100 København Ø, Denmark

Accepted February 9, 1996

Operational Reduction Models for Functional Programming Languages.  
Ph.D. dissertation, revised version 2.12 of May 20, 1997.

Copyright © 1992-1996, Kristoffer Høgsbro Rose, all rights reserved.

Typeset with the  $\text{T}_{\text{E}}\text{X}$  typesetting system (Knuth 1984), using the  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$  format (Lamport 1994), with several essential packages, notably  $\mathcal{A}\mathcal{M}\mathcal{S}$ - $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$  (American Mathematical Society 1995) and  $\text{X}\text{Y-pic}$  (Rose and Moore 1995).

# Preface

**Gentle Reader,** the present report is submitted as part of the requirements for obtaining “Ph.D.-graden ved Københavns Universitets Naturvidenskabelige Fakultet” to be presented to the public and defended 14.15 on Friday, February 9, 1996, in Lille Auditorium, Datalogisk Institut, Københavns Universitet (DIKU), Universitetsparken 1, 2100 København Ø, Danmark.

The **THESIS**, section 1.1, summarises the claimed *scientific contributions* concisely; the remainder of chapter 1 explains the themes and why we find the study undertaken in the subsequent chapters interesting and significant. This report is intended to be self-contained for someone with a background in computer science and a certain amount of mathematical maturity. To make this as wide as possible, chapter 2 is devoted to brief presentations of the preliminaries of the subject areas combined in this thesis whereas the main chapters include few definitions and explanations of such background.

**Acknowledgements.** First and foremost, I wish to thank Jan Willem Klop not only for the effort of reading this work but also for being a major force in establishing and clarifying the essential concepts of first- as well as higher-order rewriting (both to me personally and to the community at large), a feat without which this thesis could not have been.

Warm thanks also go to Peter Sestoft for evaluating and valuably commenting this work as well as contributing essentially to a sound terminology for lazy evaluation and abstract machines that has made the task of writing this thesis much easier.

And of course I could not have gotten as far as this without the continued calm perspective of my supervisor, Neil Jones, who kept insisting on the explicit formulation of relevance and purpose of the thesis even in times where such high goals seemed inconsistent with the author’s state of mind.

The quality of the thesis would surely have suffered severely had it not been

for the healthy scepticism of Roel Bloo that not only resulted in the rewarding collaboration on *preservation of strong normalisation* for named calculi presented in the first sections of chapter 3, but prompted me to clarify numerous imprecise concepts throughout the thesis.

A special thanks goes to John Hughes, who hosted me for a very pleasant and rewarding year at Department of Computer Science at Chalmers. Here I learned how pure functional programming can be used seriously, and I thank John and Thomas Johnsson for sharing their graph reduction insight with me – without it I would surely not appreciate the complexity of evaluation of pure functional programs as the subject deserves. Sincere thanks go to the individuals Zena Ariola, Henk Barendregt, Pierre Lescanne, Luc Maranget, Torben Mogensen, Rob Nederpelt, Simon Peyton Jones, Rinus Plasmeijer, John Reynolds, Mary Sheeran, Vincent van Oostrom, Femke van Raamsdonk, and several anonymous referees, for providing and provoking key insights at crucial moments; I hope you will find that I have exploited them as they deserve!

I am grateful to the following organisations and programmes for providing funding for travel: Australian Research Council, Det Danske Forskningsråd, Datalogi ved Chalmers Tekniska Högskola/Göteborgs Universitet, Department of Computing Science of Nordisk Forskerakademi, Statens Naturvidenskabelige Forskningsråd's DART (Design And Reasoning about Tools) programme, and Torkil Holms Fond, and to the following for supporting my visiting: Carnegie Mellon University, Centrum der Wiskunde en Informatica, Katholieke Universiteit Nijmegen, Macquarie University, Oregon Graduate Institute, Technische Universiteit Eindhoven, and University of Oregon (Eugene).

A special huge thank goes to Ross Moore for the continued collaboration on the diagram macro package *X<sub>Y</sub>-pic*, an enterprise that has been even more rewarding than time consuming, and without which there would only be primitive pictures in this thesis!

And I have not forgotten uncountable pleasant discussions in Olivier Danvy's usually tiny office usually in the late afternoon where many intuitions about the themes present in this thesis were helped out of their original usually uninformed state.

Finally, I am grateful to my wife, Eva Rose, for pleasant collaboration on complexity of abstract machines with cyclic data structures, her ability to always extract the essential, and for sharp comments and even for being nice to me when least reasonable these last months.

*Kristoffer Høgsbro Rose  
København, February 1996*

# Contents

Preface	3
Contents	5
List of Figures	7
<b>1 Introduction</b>	<b>9</b>
1.1 Thesis . . . . .	9
1.2 Themes . . . . .	13
1.3 Motivations and History . . . . .	16
1.4 Overview . . . . .	27
<b>2 Preliminaries</b>	<b>29</b>
2.1 Reductions . . . . .	30
2.2 Inductive Notations . . . . .	36
2.3 $\lambda$ -Calculus . . . . .	41
2.4 Strategies for $\lambda$ -Calculus . . . . .	44
2.5 Namefree $\lambda$ -Calculus . . . . .	50
2.6 Combinatory Reduction Systems (CRS) . . . . .	51
2.7 Summary. . . . .	60
<b>3 Explicit Conservative Extensions of <math>\lambda</math>-calculus</b>	<b>61</b>
3.1 Explicit Substitution ( $\lambda x$ ) . . . . .	63
3.2 Preservation of Strong Normalisation (PSN) . . . . .	75
3.3 Explicit Substitution & Naming . . . . .	83
3.4 $\lambda$ -graphs and Explicit Sharing . . . . .	96
3.5 Explicit Substitution & Sharing . . . . .	107
3.6 Summary . . . . .	112

<b>4</b>	<b>Machines for <math>\lambda</math>-calculus</b>	<b>113</b>
4.1	Abstract Machines . . . . .	114
4.2	A Call-by-Name Abstract Machine . . . . .	117
4.3	Towards a Call-by-Value Abstract Machine . . . . .	124
4.4	Abstract Machines with Sharing . . . . .	129
4.5	Summary . . . . .	132
<b>5</b>	<b>Operational Combinatory Reduction Models</b>	<b>133</b>
5.1	Explicit Substitutes . . . . .	134
5.2	Explicit Addresses . . . . .	142
5.3	CRS and Functional Programs . . . . .	146
5.4	Summary . . . . .	152
<b>6</b>	<b>Implementation of Combinatory Reduction Systems</b>	<b>155</b>
6.1	Idioms . . . . .	156
6.2	Datatypes . . . . .	159
6.3	Input . . . . .	165
6.4	Reduction . . . . .	174
6.5	Strategies . . . . .	182
6.6	Analysis . . . . .	186
6.7	Explicification . . . . .	192
6.8	Inferences . . . . .	197
6.9	Main Program & Bootstrap . . . . .	199
6.10	Summary . . . . .	204
<b>7</b>	<b>Conclusions</b>	<b>205</b>
<b>A</b>	<b>Appendix</b>	<b>207</b>
A.1	Running the CRS interpreter . . . . .	207
A.2	$\lambda$ -calculus . . . . .	212
A.3	PCF . . . . .	217
A.4	Pretty-printing Metaterms with Redexes . . . . .	225
	<b>Bibliography</b>	<b>233</b>
	<b>Index</b>	<b>245</b>

# List of Figures

1.1	Tree for 6th Fibonacci number. . . . .	23
1.2	Graph for 6th Fibonacci number. . . . .	23
1.3	Cyclic data structure representations. . . . .	24
2.1	Common $\lambda$ -calculus strategies. . . . .	45
2.2	Symbolic differentiation CRS. . . . .	59
3.1	Explicitness dimensions and selected calculi. . . . .	62
3.2	$\lambda xgc$ -reduction graph for $(\lambda y.(\lambda z.z)y)x$ . . . . .	68
3.3	Simplified version of Mellès's counterexample to PSN. . . . .	76
3.4	Garbage-free $\lambda x\downarrow gc$ -reduction graph for $(\lambda y.(\lambda z.z)y)x$ . . . . .	79
3.5	A reduction in $\lambda v$ (left) and $\lambda x$ (right). . . . .	84
3.6	$\lambda v$ . . . . .	85
3.7	$\lambda s$ . . . . .	92
3.8	$\lambda \chi$ . . . . .	94
3.9	$\lambda \sigma$ . . . . .	95
3.10	Example Wadsworth $\lambda$ -graph. . . . .	104
3.11	The Call-by-Need calculus, $\lambda_{let}$ . . . . .	110
4.1	Call-by-Name explicit substitution calculus: $\lambda x_N$ . . . . .	118
4.2	Abstract $\lambda x_N$ -machine: $\lambda x_N M$ . . . . .	122
4.3	Call-by-Value explicit substitution calculus: $\lambda x_V$ . . . . .	126
4.4	Abstract $\lambda x_V$ -machine: $\lambda x_V M$ . . . . .	128
4.5	Sharing Call-by-Name explicit substitution calculus: $\lambda x_{a_N}$ . . . . .	130
4.6	Abstract $\lambda x_N$ -machine with sharing: $x_{a_N} M$ . . . . .	131
5.1	Explicification of CRS R into ESCRS Rx. . . . .	138
5.2	Plotkin's PCF+pairs as CRS. . . . .	148
5.3	PCF+pairs with cyclic sharing as CRSar. . . . .	150

5.4	PCF <sub>+</sub> pairs as CRSar after explicification of substitutions. . . . .	151
A.1	Help message from CRS interpreter. . . . .	209



# 1

## Introduction

We commence by presenting our THESIS as section 1.1. Following, we explain the words of the title in general computer science terms in section 1.2, and give some general history of related work and motivating remarks in section 1.3. Finally, in section 1.4, we give a brief roadmap to the dissertation.

### 1.1 Thesis

WE OBSERVE that functional programming languages are very *high-level* languages in that they facilitate mathematical reasoning about the *meaning* of programs at a high abstraction level.

WE OBSERVE that usual studies of *operational* aspects of functional programs are very *low-level* in the sense that reasoning in an operationally faithful way, *e.g.*, including space or time resources requirements, requires knowledge about the used implementation technology, including knowledge of the applied *memory model* and *evaluation strategy* (including *optimisations* made before evaluating).

WE CLAIM that the gap between these two observations is an artifact of the gap between the two main models of computation: Church's (1936) ' $\lambda$ -calculus' and Turing's (1936) 'machines' because of the difference in the *size of the computation step* when compared to real computers: in general, elementary steps of

real-world computers are smaller than  $\lambda$ -steps and larger than Turing-steps.

WE ADVOCATE that a solution to this problem is found through studies of the following three paradigms:

- A. *Explicitness*: Syntactically conservative extensions that, essentially, refine the view of Church just sufficiently for the notion of computation step to be faithful to what a computer can do with respect to a particular aspect of the complexity of mechanical evaluation, *e.g.*, *sharing of computation*.
- B. *Strategies*: Restrictions on the ‘normal’ arbitrary reductions that make implementation more efficient, *e.g.*, *Call-by-Name* reduction.
- C. *Definable extensions*: Extra constructions that make it possible to realise powerful language constructions for data, operations, *etc.*, that cannot be effectively implemented in the other paradigms, *e.g.*, *explicit recursion*.

WE OBSERVE that all three paradigms seem to begin with the  $\lambda$ -calculus and can be combined orthogonally, *e.g.*, sharing, call-by-name, and explicit recursion combine to form *lazy evaluation*. We develop these paradigms in turn, treating for each its combination with the previous ones.

### 1.1.1 Contributions. Chapter 3. Explicit Conservative Extensions of $\lambda$ -calculus.

- A. WE PRESENT a naïve *explicit substitution calculus*,  $\lambda x$ , incorporating only the minimal substitution definition of Curry and Feys (1958) and the variable convention of Barendregt (1984), and we prove that  $\lambda x$  is a *conservative extension* of the  $\lambda$ -calculus [section 3.1].
- B. WE PROVE that  $\lambda x$  *preserves strong normalisation* (PSN) of  $\lambda$ -calculus. Hence using  $\lambda x$  gives finite shortest and longest reduction sequences exactly as  $\lambda$ -calculus (we give a *direct proof* in contrast to previously published proofs). Conversely we show how even the most modest extension towards composition of substitution break PSN, essentially providing much simplified versions of Melliès’s (1995) counterexample to PSN for  $\lambda\sigma$  [section 3.2, joint research with Roel Bloo].
- C. WE SYNTHESISE from  $\lambda x$  and de Bruijn’s (1972) *namefree* notation for  $\lambda$ -terms several published calculi, essentially factoring them into the explicit substitution component  $\lambda x$  and an explicit naming component. Specifically we describe  $\lambda\nu$  of Lescanne (1994a),  $\lambda s$  of Kamareddine and Ríos (1995),

and  $\lambda\chi$  of Lescanne and Rouyer-Degli (1995) this way; we can show directly that this factoring happens in such a way that PSN is maintained. This gives new *direct proofs* for preservation of strong normalisation for these calculi (existing proofs are highly indirect); the modularity of the proofs demonstrate that *explicit naming and substitution are orthogonal concerns*. Similarly we demonstrate how  $\lambda\sigma$  of Abadi, Cardelli, Curien and Lévy (1991) factors into an extension of  $\lambda\chi$  which does not possess PSN [section 3.3].

- D. WE ANALYSE *explicit sharing* as present in Wadsworth's (1971)  $\lambda$ -graph reduction, and present an abstract form of sharing based on *addressing* which we use to redefine  $\lambda$ -graph rewriting to use addresses,  $\lambda a$ , with which we can expose the main difficulties of Wadsworth's approach [section 3.4].
- E. WE SYNTHESISE explicit substitution and explicit sharing: the resulting  $\lambda\chi a$ -calculus makes reasoning about sharing possible. In particular its complexity is *faithful to  $\lambda$ -calculus reduction algorithms* in that the length of reduction mimics the best that current  $\lambda$ -calculus implementations can do [section 3.5].

### 1.1.2 Contributions. Chapter 4. Machines for $\lambda$ -calculus.

WE OBSERVE that mechanical evaluation of  $\lambda$ -terms in the tradition of Landin (1964) is most often expressed in terms of a *reduction strategy* that dictates the sequence of contractions.

WE OBSERVE that program transformation of  $\lambda$ -terms usually correspond to reductions in  $\lambda$ -calculus but rarely to reductions using the chosen strategy of evaluation.

WE CLAIM that this is remedied by studying the behaviour of reduction strategies more abstractly:

- A. WE SHOW how the generalisation of Plotkin's (1975) Call-by-Name reduction strategy for  $\lambda$ -calculus to  $\lambda\chi$  (on closed terms) gives a system that can be *systematically reduced, in a way preserving complexity*, to a generic  $\lambda$ -calculus machine (the machine is similar to published 'normal order' machines) [section 4.2].

- B. WE SHOW how a localised representation of sharing is possible within this strategy and gives rise in the same way to a *generic  $\lambda$ -graph reduction machine* (again the machine is similar to published (acyclic) ‘lazy’ machines) [section 4.4].
- C. WE OUTLINE the similar derivation of a machine for Call-by-Value reduction.

### 1.1.3 Contributions. Chapter 5. Operational Combinatory Reduction Models.

WE OBSERVE that Klop’s (1980) *combinatory reduction systems* (CRS) constitute a generalisation of the essential properties of *functional programming*, however, the computation steps of CRS correspond in complexity to the computation steps of  $\lambda$ -calculus.

WE CLAIM that functional programs (in the form of recursive equations) can be studied with the same amount of operational detail as for  $\lambda$ -calculus, by applying the previous techniques:

- A. WE IDENTIFY what problems exist with CRS reduction before the number of steps in a reduction can be said to be a complexity measure.
- B. WE SYNTHESISE *explicit substitution CRS*, generalising the  $\lambda$ -calculus notion to make it possible to obtain an explicit substitution version of any functional program.
- C. WE SYNTHESISE a notion of *explicit sharing CRS* that also captures *cyclic structures*.
- D. WE SYNTHESISE *explicit substitution and sharing CRS* from the above to provide a computation model that allows observation of *operational behaviour of functional programs*.

### 1.1.4 Contributions. Chapter 6. Implementation of Combinatory Reduction System.

WE DEMONSTRATE that the claims of operational faithfulness made in the previous section is valid by programming CRS reduction and analysing the behaviour for the subclass which is claimed operationally faithful to computers. We give several larger examples of reduction in the appendix.

## 1.2 Themes

In this section we place the dissertation in perspective as a study within computer science by giving, backwards, an explanation of each of the themes of the dissertation as manifested by the words in the title, “Operational Reduction Models of Functional Programming Languages” (the dictionary entries are taken from Collins 1979).

### “Languages”

**language** (læŋgwɪdʒ) *n.* 1. a system for the expression of thoughts, feelings, etc., by the use of spoken sounds or conventional symbols. 2. the faculty of the use of such systems, which is a distinguishing characteristic of man as compared with other animals. 3. the language of a particular nation or people: *the French language*. 4. any other systematic or nonsystematic means of communication, such as gesture or animal sounds: *the language of love*. 5. the specialised vocabulary used by a particular group: *medical language*. 6. a particular manner of style of verbal expression: *your language is disgusting*. 7. *Computer technol.* See **programming language**. 8. *Linguistics.* another word for **langue**.

In general terms a *language* is a means of communication between individuals. Languages are highly specialised to be useful for this purpose, and have gained structure that means that we learn them quickly and efficiently as children through a purely oral and social process as the necessary first step towards gaining sentience. However, otherwise this process is still poorly understood except that it is helpful when learning new languages as an adult to use a description of the language *grammar* split into *syntax*, describing how to form utterances, and *semantics*, describing what meaning is conveyed.

In computer science the word ‘language’ is used in a much more mundane way, since we only deal with languages that have been designed – at least this dissertation is concerned with *artificial languages*<sup>1</sup> only, except, of course, that it is itself written in English.<sup>2</sup> The only relic of natural languages is that we still separate the (for us mathematical) descriptions of syntax and semantics.

We summarise the notations used for syntax in section 2.2; for semantics we will use rewriting as discussed below.

---

<sup>1</sup>When a distinction is needed, real languages are often referred to as *natural languages*.

<sup>2</sup>We hope that the author’s Danish mother tongue is not too apparent.

## “Programming”

**programming language** *n.* a language system by which instructions to a computer are put into a coded form, using a well-defined set of characters that is mutually comprehensible to user and computer. See also **FORTRAN**, **ALGOL**, **COBOL**, **PL/I**, **machine language**.

In order to solve some problem with a computer, two things must be done. First the problem should be so precisely understood that it is possible to give an *algorithm* that solves the problem as a sequence of small steps that are so obvious that a computer can perform them. *Programming* is the activity of formulating the algorithm in such terms that a computer can understand it; usually this entails *implementing* the algorithm in a particular *programming language* designed for the dual purpose of being on the one hand sufficiently expressive and readable for the human programmer’s use, and on the other hand sufficiently concise and primitive that the computer will, in fact, execute the intended algorithm. Programming languages are normally categorised as *high-level* or *low-level* depending on whether the first or second part of this dual purpose is given priority, however, all programming languages satisfy both.

Since the seminal ALGOL (Naur et al. 1960), programming languages are invariably specified through a grammar<sup>3</sup> consisting of an inductive definition of the syntax of legal ‘utterances’ (programs) combined with a semantics that assigns to each such program its ‘meaning’ (what it computes).

For an introduction to programming languages in general we refer the reader to any of the immense number of introductory texts on the subject (those we recommend will be mentioned below).

## “Functional”

**functional** (ˈfʌŋkʃənəl) *adj.* 1. of, involving, or containing a function or functions. 2. practical rather than decorative; utilitarian: *functional architecture*. 3. capable of functioning; working. 4. *Psychol.* a. relating to the purpose or context of a behaviour. b. denoting a disorder without structural change. ~ *n.* 5. *Maths.* a function whose domain is a set of functions and whose range is another set of functions that can be a set of numbers.

In computer science, *functional programming languages* are high-level programming languages where the *function definition* notion is the chief means of structuring programs. The idea behind this is the thesis of Church (1936): that

---

<sup>3</sup>Hence the “ grammar = syntax + semantics ” paradigm of linguistics is maintained.

any *mechanically computable* function can be realised as a term in a particular formalism, the  $\lambda$ -*calculus*, combined with the insight of Kleene and Gödel in the 1930s that any  $\lambda$ -term can be expressed as a collection of *recursive equations* defining (mutually) *recursive functions*. For a general introduction to modern pure functional programming we recommend the introductory texts of Bird and Wadler (1988) and Hudak and Fasel (1992).

## “Models”

**model** (ˈmɒdəl) *n.* 1. a. a representation, usually on a smaller scale, of a device, structure, etc. b. (*as modifier*): *a model train*. 2. a. a standard to be imitated: *she was my model for good scholarship*. b. (*as modifier*): *a model wife*. 3. a representative form, style, or pattern. 4. a person who poses for a sculptor, painter, or photographer. 5. a person who wears clothes to display them to prospective buyers; mannequin. 6. a preparatory sculpture in clay, wax, etc., from which the finished work is copied. 7. a design or style, esp. one of a series of designs of a particular product: *last year’s model*. 8. *Brit.* a. an original unique article of clothing. b. (*as modifier*): *a model coat*. 9. a simplified representation or description of a system or complex entity, esp. one designed to facilitate calculations and predictions.

A *model* is a mathematical structure that captures essential aspects which we wish to *observe* in a concrete system. A model of a programming language is *correct* (or *sound*) if it captures the *value* of computations without error: if two programs can compute different values then they should correspond to different objects in the model. Dually, models can be *complete*: this is the case when an observable difference in the model means that the corresponding programs are ‘really different’. Models that are both sound and complete are called *fully abstract*. We shall not be concerned with completeness in this dissertation, since our chief concern is to *enrich* models to make particular aspects of the *computation process* observable.

## “Reduction”

**reduction** (rɪˈdʌkʃən) *n.* 1. the act or process of an instance of reducing. 2. the state or condition of being reduced. 3. the amount by which something is reduced. 4. a form of an original resulting from a reducing process, such as a copy on a smaller scale. 5. a simplified form, such as an orchestral score arranged for piano. 6. *Maths.* a. the process of converting a fraction into decimal form. b. the process of dividing out the common factors in the numerator and denominator of a fraction; cancellation.

As the example above hinted, we will model program evaluation by rewriting. Systems defining such rewriting are usually called *reduction systems*. A formal description of the reduction system we used to describe the implementation of `fib` above would be a reduction model for the `fib` program. In general, we can make a reduction model for a functional programming language by associating to each actual program a reduction system that evaluates the program. This is what we will do in this dissertation. We summarise reduction systems in section 2.1.

## “Operational”

**operation** (ɒpəˈreɪʃən) *n.* 1. the act, process, or manner of operating. 2. the state of being in effect, in action, operative (esp. in the phrases *in or into operation*). 3. a process, method, or series of acts, esp. of a practical or mechanical nature. 4. *Surgery.* any manipulation or one of its organs or parts to repair damage, arrest the progress of disease, remove foreign matter, etc. 5. **a.** a military or naval action, such as a campaign, manoeuvre, etc. **b.** (*cap. and prenominal when part of a name*): *Operation Crossbow*. 6. *Maths.* any procedure, such as addition, multiplication, involution, or differentiation, in which one or more numbers or quantities are operated upon according to specific rules. 7. a commercial or financial transaction.

**operational** (ɒpəˈreɪʃənəl) *adj.* 1. of or relating to an operation or operations. 2. in working order and ready for use. 3. *Military.* capable of, needed in, or actually involved in operations.

Something is operational when it can be run on a computer. This means that in this dissertation we concentrate on the *algorithms*, using Plotkin’s (1981) definition of *operational*: “intuitively corresponding to computation steps . . .”. This is narrower than some other definitions of operational semantics in that we insist that the computation steps observable in the model should be mappable onto a real computer. We will, of course, want to *compare* the models we obtain, therefore we will keep strictly to reduction systems over inductively defined sets.

## 1.3 Motivations and History

We restate our thesis motivations and then qualify it by discussing several historic issues.



## Motivations

Pure functional programming is spreading . . . mostly indirectly through a growing understanding of the advantages of its principal feature, *referential transparency*; this is useful for example for ensuring modularity of large programming systems (we give an example below).. Indeed even purely functional programming *languages* are becoming more widely used.

Computationally these languages are commonly understood through a mapping to Church's (1936)  $\lambda$ -calculus model of the computable functions (we return to this below). This is a very abstract formalism, and the association has introduced a major obstacle for functional languages, namely the lack of intuition of *operational properties*, such as time and space complexity, of the execution of functional programs. For this reason research in functional programming languages has tended to focus on either very 'high-level' issues such as potent program transformation (because it is possible!) or on correspondingly 'low-level' problems such as compilation into efficient machine code (because it is difficult!). And consequently it is very difficult to reason abstractly about the time and space complexity of algorithms when they are expressed using a functional programming paradigm.

In order to attack this problem we will briefly outline why we consider functional programming nice, and touch on some aspects of what a computer is, how its behaviour is analysed through complexity considerations, why this is difficult for in particular *sharing* considerations, and finally mention some means used to describe these issues in the past.

## Why functional programming?

It is usually claimed that the origins of functional programming is the  $\lambda$ -calculus but Church's interest was in the notion of computable functions (on numbers) induced by  $\lambda$ -definability (to be explained below). However, the related notion of *recursive equations* was always a popular specification notation so when McCarthy (1960) showed that recursive equations could be directly implemented using a  $\lambda$ -calculus *evaluator*, the resulting new style of programming quickly became popular in the computer science community.

In recent years, *pure functional programming languages* have shifted from being mere scientific toys to become a serious basis for implementations of large systems. Or rather: they have had a (quiet) impact on standards and extensions of (non-functional) languages, protocols, *etc*, because of the in our opinion main

feature of pure functional programming languages: *referential transparency*. An example of this is the requests of the HTTP protocol (Berners-Lee, Fielding and Frystyk 1995) for the *world wide web* (Cailliau 1995) which is *stateless* in the sense that each request contains full context information. This has been quoted as an essential property for making HTTP realisable on a large scale because *servers do not need to keep track of active connections*.

However, while the advantages of statelessness seem obvious, the disadvantages are equally obvious. Using the example of HTTP, one disadvantage is that the size of requests is sometimes larger than the answer (this is not noticeable at present because the unit of network communication is quite large, typically  $2^{14}$  bits – but the communication unit is destined to become much smaller –  $2^6$  bits – in the near future). The lack of state can be ‘solved’: it is possible to encode state into a stateless context. For the HTTP protocol this is demonstrated by the fact that servers *can* allow encoding of ‘state identifiers’ into the requests. For example, a popular device is to let some server keep a count of network accesses to a particular data item: this is achieved by inventing an identifier and passing this identifier along with each request to the counter server, *e.g.*, a document by the author includes the URL<sup>4</sup> `http://www.forsmark.uu.se/cgi-bin/ccounter?xypic&width=4` where *xypic* is a state name. The important insight provided by this is, of course, that *you cannot get rid of state* in specifications because one cannot predict what a connection will be used for in the future – if the HTTP specification had included a proviso for state then *every* server would have to suffer from the overhead of keeping track of all connections because at some point in the future they just *might* make use of it.

This example illustrates very concretely the problem facing large programming systems: programs are made by people who tend to build components that are interdependent to the extent that the programmer can understand it. This means that small modifications involving only the domain of a few programmers are easy to realise whereas large modifications that involve changes to code produced by a large number of programmers are difficult to realise.<sup>5</sup> Expressing algorithms in a functional programming language requires that all interdependencies between components is *explicit* so in a way one can say that functional programming plays the same rôle as standards such as HTTP but on a smaller scale, even within the work of a single programmer. This means that the individual components of a single programmer can more easily be exchanged.

---

<sup>4</sup>URL means Uniform Resource Locator, the user configurable part of an HTTP request.

<sup>5</sup>By ‘difficult’ we mean that large modifications involve more work *per programmer*, of course.

## What is a computer?

Computers were originally developed following the principles of von Neumann and these are the principles still used in virtually all computers today because they are so simple and hence reliable. Basically, a computer is always in a *configuration* consisting of a large number of ‘bits’ (up to  $2^{40}$ ), each with a unique address. The computer then repeatedly cycles through an *instruction cycle* where a designated small number of bits (typically  $2^{10}$ ) are investigated by the *central processing unit* to affect a change of the configuration according to specific rules that can change most of these designated bits and a small number of the remaining configuration bits. After the cycle the designated bits are ready for the next instruction (current computers perform up to  $10^9$  instructions per second).

The fact that only a small portion of the configuration can have direct influence on the following configuration was dubbed the “von Neuman bottleneck” by Backus (1978): the fact that computation models do not distinguish between ‘close’ and ‘distant’ data is not in accord with the fact that the instruction cycle is realised by a single electronic circuit which is subject to the natural limitation that electronic signals cannot travel faster than the speed of light. This is fortified by the fact that the configuration is structured further into a hierarchy: Typically a small number of  $2^{20}$  designated ‘secondary’ bits (the ‘cache’) can be accessed within a few instruction cycles. Accessing any of the  $2^{30}$  ‘tertiary’ bits (the ‘internal memory’) can only be done indirectly by associating them to specific secondary bits which takes at least 10 times longer, and accessing any of the general  $2^{40}$  configuration bits (the ‘external memory’ or ‘disk pool’ connected together in local area networks) takes at least 1000 times longer than that since they need to be accessed by being addressed via tertiary memory (and depend on the speed of mechanical devices). If the entire ‘world net’ is included in the configuration then accessing a small number of the world’s  $2^{60}$  bits is 1000 times slower again (the numbers should only be seen as indicative of the magnitudes). These magnitudes may seem staggering, however, it is a fact that today’s state of the art, in fact, makes access to any bit properly connected to the world net possible in under a second!

This structure means that *locality* of access to data and program components is important, and is something that should be included in considerations of how to construct programs – in fact the technology of contemporary computers support functional programming paradigm in that forcing the programmer to make interdependencies between program components explicit means that there

will be fewer interdependencies which again means that each component is more likely to be able to run without too many references to ‘far’ contexts! This is not easy to get right, however, and is a major reason for the considerable body of work on functional language implementation on parallel computers. In this thesis we will include locality as an essential consideration.

## Computability and complexity

In the thirties computable functions (on numbers) were studied abstractly, and Gödel (1931) found that not all functions were computable. Concurrently three refined views were found: Church (1936) developed the  $\lambda$ -calculus and claimed computability to be “ $\lambda$ -definability”; Turing (1936) developed the notion of a “machine”, and the notion of *recursive functions* was defined by Kleene (1981). These were quickly shown to define the same class of functions (Turing 1937). This supports *Church’s Thesis* which is the claim that this class in fact contains the “mechanically computable” functions.

However, that a function is computable is no guarantee that it is feasible to actually compute it! In particular the  $\lambda$ -calculus as an *operational* model of computation is not very successful: its ‘reduction unit’,  $\beta$ -reduction, is not even local! Turing’s machines are better in this respect: the notion of a ‘tape’ captures the idea of locality very well, however, Turing’s notion of memory is one-dimensional, which is not reasonable in the light of the above, and in addition the translation of functional programs into Turing machines is highly nonintuitive. In short, Turing machines do not constitute any real help in understanding in an intuitive way the complexity of execution of functional programs. In particular the complexity of sharing, described next, is hard to capture.

## Sharing

In fact, most modern programming languages have a notion of *sharing* either explicitly or implicitly. An example is the data sharing explicit in PASCAL *pointers* (Wirth 1971) (present in most contemporary imperative programming languages), LISP *location equality* (the EQ predicate of McCarthy 1960), and SML *references* (Milner, Tofte and Harper 1990). A more interesting example is the data and code sharing implicit in the *Call-by-Name* parameters of ALGOL (Naur et al. 1960).<sup>6</sup> Code sharing is also explicit in *object-oriented programming*.

---

<sup>6</sup>It is interesting to notice that, by mistake, what is specified in the ALGOL report is what is now denoted *Call-by-Text* yet the intention, and what all the ALGOL compilers used, was, in

In pure functional programming languages, sharing of both data and computation is present implicitly via lazy evaluation, and sharing of data is available through explicit data declarations. For functional programming this is the aspect most often swept under the carpet, usually in one of the following ways:

**Ignored:** “Sharing does not add to the expressive power so we ignore it, relying on ‘standard implementation techniques’ for substitution and recursion.”

**Hidden:** “We rely on the underlying logic/specification formalism to hide sharing and thus do not need to take it into account.”

**Low-level:** “Sharing is handled by modeling the computer memory in the form of a global ‘store’ component updated by each reduction.”

The first approach means that the resulting model is not operationally faithful to the described language in the sense that single reductions may need to do an unbounded amount of copying when performed by the computer. An important example of this is the *Categorical Abstract Machine*, CAM, that is beautifully derived by Cousineau, Curien and Mauny (1987) except for the recursion instruction *wind* which is basically motivated by its similarity to the *ad hoc* primitive RPLAC of LISP. For a proper treatment of this aspect see Rose (1996).

*Hiding* the issue means that even the model is only operationally faithful to the language insofar as the formalism it is based on is. An example of such a model is given in Kahn (1987) where the fact that the underlying reduction machine is PROLOG (namely that PROLOG omits the so-called “occurs check” and allows unification of a structure with a substructure of itself).

The gain of the first two approaches is, however, that the semantic descriptions can be reductions based on a *term representation* and thus are much easier to work with because of the nice and simple induction principles that term structure permits.

The *low-level* approach does give operationally faithful models but they cannot be based purely on terms because a strict distinction is necessary between “global” properties such as ‘for every node everywhere in the store ...’ and “structural” term properties. Thus the resulting models are too “low-level” in the sense that they do not provide much help to the semanticist for proving operational properties of program transformations with respect to sharing. For the

---

fact, Call-by-Name as we still define it.

purposes of this dissertation this is the problem with most approaches (including Plotkin 1981, Barendregt, van Eekelen, Glauert, Kennaway, Plasmeijer and Sleep 1987, Launchbury 1993, Jeffrey 1993).

What is needed is descriptions of sharing that make it possible to reason with the sharing for aspects that benefit from it, such as complexity, and reason without it easily whenever that is convenient. As a simple example consider the following:

**1.3.1 Definition (Fibonacci number).** The  $n$ th *Fibonacci number*,  $F_n$  is defined for any  $n \in \mathbb{N}_0$  by the cases

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2 \end{cases}$$

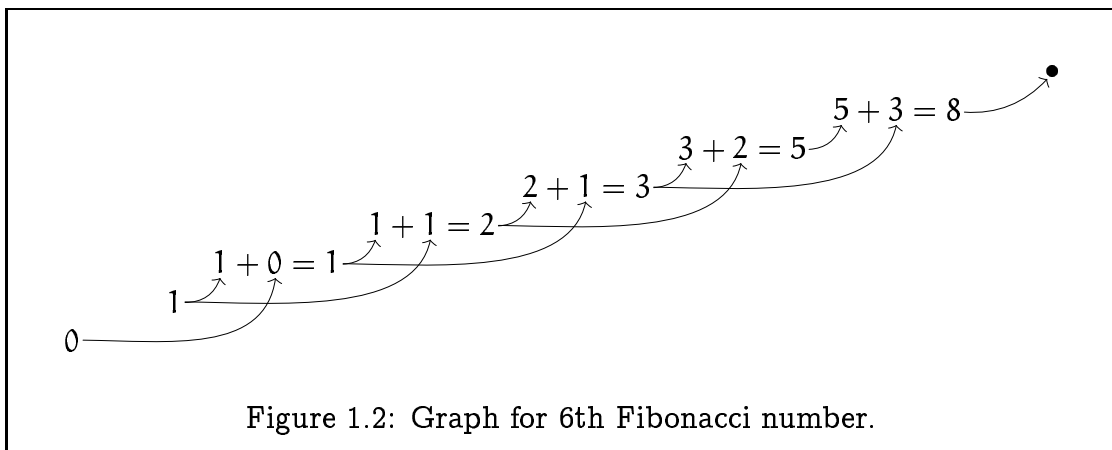
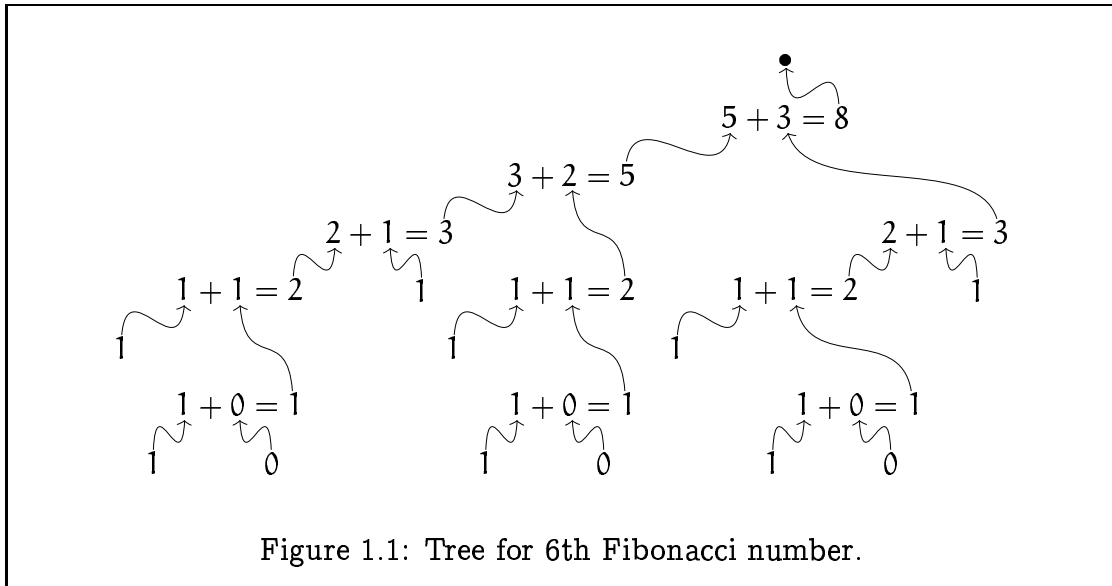
The  $F_n$  sequence was invented by Pisano (1209) as a solution to the exercise “How many pairs of rabbits can be reproduced from a single pair in a year’s time?” with certain assumptions (see Knuth (1973, section 1.2.8) for some details including the fact that Pisano was called ‘Fibonacci’ because he was ‘son of Bonaccio’).

The definition can be seen as an algorithm but not a very efficient one, in fact Fibonacci himself wrote on mechanically following it that “It is possible to do in this order for an infinite number of months.” This is because the algorithm essentially constructs a tree, for example, computing  $F_6$  amounts to building the tree in Figure 1.1 (this is small compared to the computation of  $F_{12}$  that Fibonacci originally required).

It is apparent that this tree repeats a lot of computation. A simple analysis shows that to compute the  $n$ th Fibonacci number this way an exponential number of reductions are needed (approximately  $1.6^n$ )! Thus for  $F_6$ ,  $1.6^6 \approx 10$  computation steps are done which is rather a lot, considering that altogether only seven different values are computed, namely  $F_0, F_1, F_2, F_3, F_4, F_5$ , and  $F_6$ . The problem is, of course, that the definition as algorithm does not express the *sharing of computation* that the sensible programmer can quickly see. Graphically, we would like the algorithm to express the data flow in Figure 1.2.

The sharing becomes even more interesting when “self-sharing” is involved, *i.e.*, cyclic data and function dependencies. An example is the data structure in Figure 1.3.

The left picture represents the data structure abstractly as a graph; in the right picture we have chosen to *order* the components of the graph in a linear way



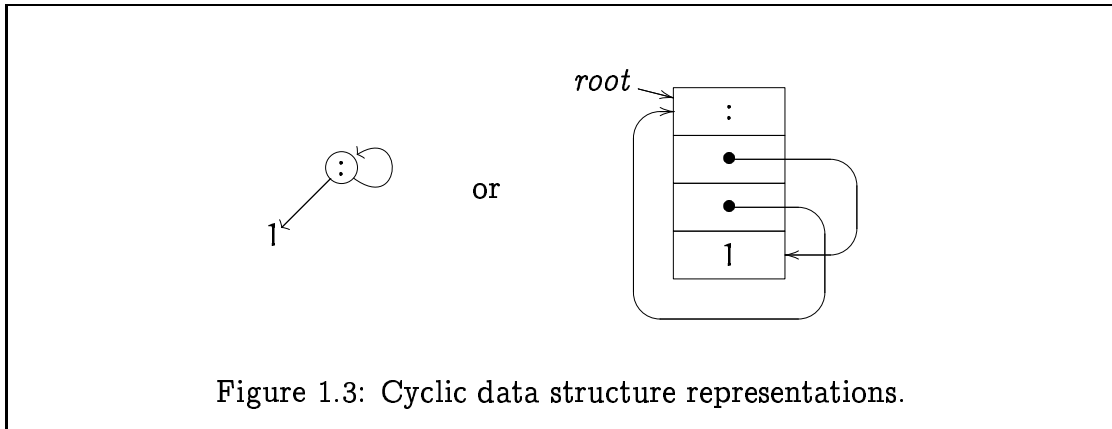
in order to show the similarity with a computer memory representation of the structure. But this is really entering into the next subject ...

## Functional Programming

How would the above be implemented in functional programming languages?

The following example implements the Fibonacci computation of Definition 1.3.1 as a simple program with one declaration containing three cases that are mutually exclusive.<sup>7</sup>

<sup>7</sup>The example also illustrates how we will show programs with little numbers in front of each line because they are, in fact, executable fragments in what is known as *literate Haskell*, the



**1.3.2 Code (Fibonacci number).** `fib n` computes the *n*th Fibonacci number.

```

1 fib 0      = 0
2 fib 1      = 1
3 fib (n+2) = fib(n+1) + fib n
? fib 6

```

To evaluate this program we require that the specified application, `fib 6`, matches exactly one of the cases: the last. Progress towards the final result, *simplification*, is obtained through *rewriting* the expression according to the appropriate case (oriented from left to right). For `fib 6` this means realising that the last equation is the only one that matches, since the binding  $n = 4$  is possible. This results in the rewriting

$$\text{fib } 6 \longrightarrow \text{fib}(4+1) + \text{fib } 4$$

Say we wish to rewrite the first of the addends. Before we can match it against one of the equations we need to know whether the argument is 0, 1, or something else, hence the next simplification is

$$\text{fib}(\underline{4+1}) + \text{fib } 4 \longrightarrow \text{fib } 5 + \text{fib } 4$$

where we have underlined the subterm that has been reduced. Now we can use the last equation again obtaining

$$\underline{\text{fib } 5} + \text{fib } 4 \longrightarrow (\text{fib}(3+1) + \text{fib } 3) + \text{fib } 4$$

and so on, until we finally reach

---

style we also use for real programs, see Notation 6.1.1.



$$\underline{\text{fib } 1} + \dots \longrightarrow 1 + \dots$$

where the second equation is applicable for the shown component; similarly for the remaining ones. If the simplification of  $+$  is delayed as much as possible then the last rewrites will be

$$((((((1+0)+1)+(1+0))+((1+0)+1)))+(((1+0)+1)+(1+0))) \longrightarrow 8$$

where each 0 is the result of `fib 0` and each 1 the result of `fib 1` (corresponding to the tree of Figure 1.1) altogether this yields the correct value of the sixth Fibonacci number.

This highlights an essential property of *pure* functional programs is that the exact sequence we evaluate in should not matter for the result. This means that we can evaluate any way we like as long as the evaluation is guaranteed to find the result if there is one (so we cannot use CBV evaluation, for example). While one may be told that a particular pure functional language evaluates uses a particular evaluation strategy, say the *lazy* strategy that we discuss in this thesis, this is usually not true since any serious pure functional programming language *implementation* will do many so-called *program transformations* which are nothing but reductions that do *not* obey the standard evaluation principle.

Finally, as discussed in the introduction, algorithms that improve the *sharing* are often searched for. For example, we can realise a sharing version of Fibonacci by the following alternative algorithm that computes *pairs* of values:

**1.3.3 Code (Fibonacci number, fast).** From (Bird and Wadler 1988, section 5.4.5).

```

1 fastfib n = fst(twofib n)
2 where fst(n,_) = n
3     twofib 0    = (0,1)
4     twofib (n+1) = (b,a+b) where (a,b) = twofib n

```

The trick used is that the last two computed numbers in the sequence are “memorised” (as  $a$  and  $b$  by the local definition) such that they can be used to compute the next in the sequence before the oldest is discarded and the process repeated. The overall complexity of this is linear (corresponding to the size of the graph in Figure 1.2).

Finally we mention how infinite data structures are coded directly using *recursion*.

**1.3.4 Code (infinite lists).**

```

1 ns n = ns' where ns' = n : ns'

```

This declaration defines a function `ns` that builds an infinite list of copies of its argument. For example, the expression `ns 1` will build a cyclic data structure like the one in Figure 1.3.

## Graph Reduction

It is interesting to see how graph representations like those above have quietly become part of the “folklore” of implementation technique for functional programming languages during the last three decades. It all started with the LISP programming system: McCarthy (1960) writes “it is permitted for a substructure to occur in more than one place in a list structure, [ . . . ], but it is not permitted for a structure to have cycles”. Thus, “pure LISP” essentially allows only acyclic sharing in the execution model. The effect of cycles must be constructed by rewriting through use of the LABEL form that defines a recursive function. However, all realistic uses of “real LISP” (McCarthy, Abrahams, Edwards, Hart and Levin 1965) use the RPLAC primitive to build proper cyclic data structures (in fact they usually implement LABEL this way). Since then all implementations of functional languages have used “graph tricks” to get better sharing and avoid unnecessary symbol lookup; while at the same time their *models* have avoided the issue.

Acyclic sharing techniques were formalised early for the  $\lambda$ -calculus (as we shall discuss in section 3.4) by Wadsworth (1971) but the technique of using cyclic pointer structures for recursion has only been formalised in retrospect since implementations of functional languages using *graph reduction* like the G-machine (Johnsson 1984, Peyton Jones 1987) has become common, creating a need for optimisations. The most successful such formalisation was *term graph rewriting* (TGR) of Barendregt et al. (1987) because of the simple idea that the computer memory can be seen as a (rooted, labelled, ordered, and directed) graph by interpreting each memory address as a graph *node id*, the type of data in the word as a *label* and any pointers to other addresses as *edges* (or *arrows* since the graph is directed). Term Graph Rewriting has been used to model sharing in several types of languages (Glauert, Kennaway and Sleep 1989) as well as other aspects of functional programming language implementations (Ariola and Arvind 1992, Koopman 1990, Jeffrey 1993). These models are useful because the TGR formalism has sharing and recursion (cycles) built in, but the way they have been constructed from existing languages, in particular the way they depend in nontrivial ways on the *evaluation strategy* used by existing functional

programming languages, limits how much we can learn about the limitations of these languages. Furthermore, term graph rewriting does not contain a notion of ‘locality’ in the way discussed above: all accesses ‘point the same distance’. This gives severe problems with implementation that use ‘parallel graph rewriting’ (Plasmeijer and van Eekelen 1993). And finally, term graph rewriting is a very “low-level” description in that the operations on graphs are understood in terms of very primitive “simple” operations: the original authors used small pieces of imperative programs; the use of category theory as done by Raoult (1984) and the school of Ehrig (1978) does not really help with this aspect.

A recent generalisation of Wadsworth’s that also handles cyclic sharing is the  $\lambda$ -graph rewriting of Ariola and Klop (1994), however, the introduced formalism is even stronger than term graph rewriting (we compare to this closely related work in the thesis text).

## 1.4 Overview

This dissertation is structured in chapters, each with a separate introduction and brief summary. The order of the chapters is significant, in particular each chapter refers freely to material in earlier chapters. No detailed table of contents exist as the index should be of help when searching for concepts and named results.

As proclaimed in the THESIS, the main chapters are chapters 3, 4, and 5; Chapter 2 contains introductory material on the established theoretical fields underlying this work, and chapter 6 contains the complete source of the CRS reduction program we have used to verify that most of the syntactic transformations of reduction systems that we do throughout are right (the output and a few periferal components are in the appendix).

We conclude and present a personal perspective and some future directions in chapter 7.



# 2

## Preliminaries

This dissertation covers several computer science research areas: *rewriting* (even of *higher order*), *semantics* (of *programming languages*), *pure functional programming*, and *implementation issues* (in particular *time and space complexity*). The intention with this chapter is to make the dissertation self-contained by ensuring that the reader is sufficiently familiar with these areas to read the remaining chapters.

We first explain the reduction concept of (abstract) rewriting in section 2.1 because the notions involved are useful in explaining the remaining concepts, in particular the idea of diagrammatic propositions. We next explain in section 2.2 the inductive notations of semantics that we will use, including inductive diagrammatic propositions.

The following sections are devoted to aspects of the  $\lambda$ -calculus common to semantics and functional programming. We start with a brief summary of traditional  $\lambda$ -calculus in section 2.3, proceed with a short survey of evaluation strategies for  $\lambda$ -calculus in section 2.4, and conclude with a summary of de Bruijn's (1972) 'namefree' calculus in section 2.5.

Finally, in section 2.6 we give an introduction to *combinatory reduction systems* (CRS) of Klop (1980), a very general rewriting formalism which includes concepts from all of the above while retaining nice theoretical properties.

## 2.1 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.

### 2.1.1 Notations (relations).

- A. An *n*-ary relation ( $n \geq 2$ ) is a subset  $R \subseteq A_1 \times \dots \times A_n$  where we write  $R(a_1, \dots, a_n)$  for  $(a_1, \dots, a_n) \in R$ .
- B. 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*.
- C. 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  $\nrightarrow$ , is defined as  $A \times A \setminus \rightarrow$  (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  $\emptyset$ .

**2.1.2 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).

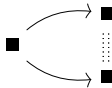


Most relations in this dissertation will be reductions. In particular we will use the common equality sign  $=$  to mean *equality* in the strictest syntactic sense available in each case.<sup>1</sup> If the base set is unclear we write  $=|_A$  (one can think

<sup>1</sup>Thus “=” is not used to denote  $\beta$ -convertibility.

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 difficult to understand by considering the following example definitions of some well-known standard properties of abstract relations.

### 2.1.3 Definitions (examples of properties as diagrams).

- A.  $\rightarrow$  is a *function*, or *deterministic*, if . A function defined over an enumerable range is a *map*.
- B.  $\rightarrow$  is *total* if , otherwise it is *partial* (but we will rarely say so).
- C. Given  $\rightarrow$ . The *converse* of  $\rightarrow$  is the relation  $\rightarrow_x$  defined by ; it is simply denoted  $\leftarrow$ .

The formal definition below translates these as follows: **A** becomes  $\forall a, b, c : (a \rightarrow b \wedge a \rightarrow c) \implies (a = b)$ , **B** becomes  $\forall a \exists b : a \rightarrow b$ , and **C** becomes  $\forall a, b : (a \rightarrow b) \implies (b \rightarrow_x a)$ . The definition of converse means that we must take care to use asymmetric symbols for asymmetric relations in diagrams.

**2.1.4 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 occurring *only* framed,  $P^\square$  is a conjunction of all relations where the relation symbol is dotted (*e.g.*,  $\dashrightarrow$  and  $\dashrightarrow_x$  are dotted versions of  $\rightarrow$ ), 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.

**2.1.5 Definitions.** Let  $\rightarrow$  be a reduction on  $A$ .

A. It is *reflexive* iff  $\blacksquare \overset{\curvearrowright}{\rightarrow} \blacksquare$ .

B. It is *transitive* iff  $\blacksquare \overset{\curvearrowright}{\rightarrow} \blacksquare \overset{\curvearrowright}{\rightarrow} \blacksquare$ .

C. It is *symmetric* iff  $\blacksquare \overset{\curvearrowright}{\rightarrow} \blacksquare$ .

D. It is *antisymmetric* iff  $\blacksquare \overset{\curvearrowright}{\rightarrow} \blacksquare$ .

E. It is an *equivalence* if it is reflexive, transitive, and symmetric (such a reduction is sometimes called a *conversion*).

F. It is a *partial order* if it is reflexive, transitive, and antisymmetric.

**2.1.6 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  $\blacksquare \xrightarrow{AB} \blacksquare \xrightarrow{BC} \blacksquare$ .

In particular  $\leftarrow \cdot \rightarrow = (=|_B)$  if  $\rightarrow$  is a total function from  $A$  to  $B$ . We will use several shorthands for reduction compositions.

**2.1.7 Notations (compositions).** Let  $\rightarrow$  be a reduction on  $A$ .

A.  $\xrightarrow{n}$ , for  $n \in \mathbb{N}_0$ , is the *n-fold composition* defined inductively by

$$\xrightarrow{n} = \begin{cases} =|_A & \text{if } n = 0 \\ \rightarrow \cdot \xrightarrow{n-1} & \text{if } n > 0 \end{cases}$$

B.  $\xrightarrow{n:m}$ , where  $n, m \in \mathbb{N}_0$ , is the union  $\xrightarrow{n} \cup \xrightarrow{n+1} \cup \dots \cup \xrightarrow{m}$  (the empty relation if  $n > m$ ). If we omit  $n$  and/or  $m$  then  $0$  and/or  $\infty$ , respectively, are understood.<sup>2</sup>

Usually one establishes a property by either extending or restricting the relation as little as possible to ensure it.

<sup>2</sup> $\infty$  just means there is no upper limit (not infinitary reduction or such).



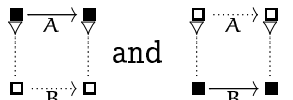
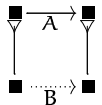
**2.1.8 Notations (closures).** Let  $\rightarrow$  be a reduction on  $A$ . For some concept  $X$ , the  $X$  *closure* of  $\rightarrow$  means the (unique) smallest reduction  $\rightsquigarrow$  on  $A$  which satisfies  $X$  and such that  $\rightsquigarrow \supseteq \rightarrow$ . In particular,

- A. The reflexive and/or transitive closures of  $\rightarrow$  are defined by composition:  
 $\xrightarrow{=} = \xrightarrow{!}$ ,  $\xrightarrow{+} = \xrightarrow{!}$ , and  $\xrightarrow{*} = \xrightarrow{!}$ . (The first and last are denoted  $\xrightarrow{\epsilon}$  and  $\xrightarrow{*}$  by Huet (1980).)
- B. We will use  $\Leftrightarrow$  for  $\rightarrow$ 's *equivalence closure*.<sup>3</sup>

**2.1.9 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.

**2.1.10 Definitions (translation).** Given  $\xrightarrow{A}$  and  $\xrightarrow{B}$  on  $A$  and  $B$ , respectively, and a relation  $\triangleright$  from  $A$  to  $B$ .

- A.  $\triangleright$  is a *translation* of  $\xrightarrow{A}$  to  $\xrightarrow{B}$  if .
- B.  $\triangleright$  is a *projection* of  $\xrightarrow{A}$  into  $\xrightarrow{B}$  if furthermore .

**2.1.11 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  $\xrightarrow{A}$ -reductions in  $A$  with a single  $\xrightarrow{B}$ -reduction in  $B$ .

Our notion of projection is closely related to, but weaker, than the notion of a *Galois connection*: if

- $\triangleright$  is a projection function of  $\xrightarrow{A}$  into  $\xrightarrow{B}$ , and
- a function  $\rightarrow$  from  $B$  to  $A$  is given such that  $\leftarrow \subseteq \triangleright$ ,

<sup>3</sup>This is *not* always obtainable by constructing the reflexive, transitive and then symmetric closures in some sequence, of course, even if the notation  $\Leftrightarrow$  might be interpreted to indicate just that.

then the pair of  $\triangleright$  and  $\rightarrow$  is a Galois connection between  $\xrightarrow{A}$  and  $\xrightarrow{B}$ . 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  $a' \in A$  through  $a \triangleright \cdot \rightarrow a'$ .

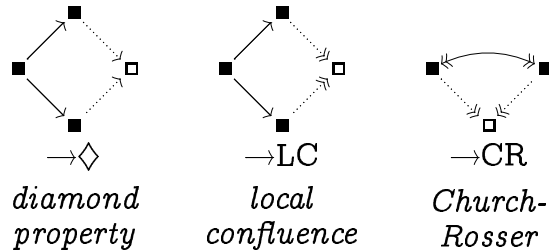
The special case of this when  $A \supseteq B$  (such that the translation  $\triangleright$  is itself a reduction on  $A$ ) is particularly simple and has a standard name for transitive closures.

**2.1.12 Definition (conservative extension).** Given reductions  $\xrightarrow{A}$  and  $\xrightarrow{B}$  on  $A$  and  $B$ , respectively, and a projection  $\triangleright$  from  $\xrightarrow{A}$  to  $\xrightarrow{B}$ . If  $A \supseteq B$  then  $\xrightarrow{A}$  is a *conservative extension* of  $\xrightarrow{B}$ , and  $\xrightarrow{B}$  is a *subreduction* of  $\xrightarrow{A}$ . 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 dissertation: confluence and normalisation.

**2.1.13 Definition (confluence).**



We say that  $\rightarrow$  is *confluent* when  $\rightarrow\Diamond$ . In Klop (1992), LC is denoted WCR (for ‘weak CR’).

A classical result is the following.

**2.1.14 Proposition.**  $\rightarrow$  is confluent if and only if  $\rightarrow CR$ .

*Proof.* The ‘if’ direction is trivial. The ‘only if’ direction is proved by assuming  $\rightarrow\Diamond$  and consider a conversion  $a \Leftarrow\!\!\!\rightarrow b$ : this must have the shape

$$a \rightarrow a_{(1)} \Leftarrow a_1 \rightarrow a_{(1,2)} \Leftarrow a_2 \cdots a_n \rightarrow a_{(n)} \Leftarrow b$$

for some  $n$ . By  $\rightarrow\Diamond$  we then have

$$a \rightarrow a_{(,1)} \rightarrow a_{((,1),(1,2))} \leftarrow a_{(1,2)} \rightarrow a_{((1,2),(2,3))} \leftarrow a_{(2,3)} \cdots \leftarrow b$$

and by induction there is a common reduct and hence  $\rightarrow\text{CR}$ . 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).  $\square$

**2.1.15 Definitions (normalisation properties).** Let  $\rightarrow$  be a relation from  $A$  to  $B$ .

- A. The *normal forms*  $\rightarrow\text{-nf}$  are those  $x \in A$  for which  $\nexists y : x \rightarrow y$ .
- B. The *final restriction*  $\rightarrow\ast$  is defined by  $w \rightarrow\ast x$  iff  $w \rightarrow x$  and  $x$  is an  $\rightarrow\text{-nf}$ .
- C.  $\rightarrow$  has *unique normal forms*, written  $\rightarrow\text{UN}$ , if  $\rightarrow\ast$  is a function.
- D. The set of  $\rightarrow$ -*weakly normalising* objects is the subset  $\text{WN}_{\rightarrow} \subseteq A$  defined by  $\text{WN}_{\rightarrow} = \{x \in A \mid \exists y : x \rightarrow\ast y\}$ . We say ‘ $\rightarrow$  is weakly normalising’, and write  $\rightarrow\text{WN}$ , when  $\text{WN}_{\rightarrow} = A$ .
- E.  $\rightsquigarrow$  is said to *preserve weak normalisation* (PWN) of  $\rightarrow$  if  $\text{WN}_{\rightsquigarrow} \supseteq \text{WN}_{\rightarrow}$ .
- F. The set of  $\rightarrow$ -*strongly normalising* objects is the subset  $\text{SN}_{\rightarrow} \subseteq A$  defined such that  $x \in \text{SN}_{\rightarrow}$  implies that all reductions  $x \rightarrow x_1 \rightarrow \cdots$  from  $x$  are finite. We say ‘ $\rightarrow$  is strongly normalising’, and write  $\rightarrow\text{SN}$ , when  $\text{SN}_{\rightarrow} = A$ .
- G.  $\rightsquigarrow$  is said to *preserve strong normalisation* (PSN) of  $\rightarrow$  if  $\text{SN}_{\rightsquigarrow} \supseteq \text{SN}_{\rightarrow}$ .
- H.  $\rightarrow$  is *complete* if it is CR and SN.

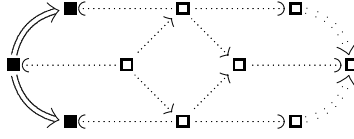
**2.1.16 History.** Huet (1980) uses *noetherian* (after Emmy Noether) for SN; sometimes *terminating* is used. Some use  $\rightarrow^!$  for what we write as  $\rightarrow\ast$ . Klop (1992) uses  $\text{UN}^{\rightarrow}$  for what we call UN.

The reasons why PSN and PWN are interesting in their own right, and the reason there is no ‘preservation of  $\Diamond$ ’, is the following.

**2.1.17 Propositions.** Given reductions  $\rightarrow$  and  $\Rightarrow$  with a translation  $\rightarrow$  from  $\rightarrow$  to  $\Rightarrow$ .

- A.  $\rightarrow \diamond$  iff  $\Rightarrow \diamond$
- B. If  $\rightarrow$  is a projection then  $\Rightarrow$ WN implies  $\rightarrow$ WN
- C. If  $\rightarrow$  is a projection then  $\rightarrow$ SN implies  $\Rightarrow$ SN

*Proofs.* A is proved by an instance of what is called the ‘interpretation method’ by Hardin; we show the ‘only if’ case (the ‘if’ is symmetric).



B follows from the fact that we *remove* reduction elements when taking a sub-reduction so no infinite ones can be added. C by the converse argument: since a conservative extension preserves reductions then it also preserves the normalising ones.  $\square$

## 2.2 Inductive Notations

This section defines the basic inductive notations and terminology used in this dissertation. 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.

### 2.2.1 Notations (sequences).

- A. A *finite sequence* over  $\Phi$ ,  $\phi = (\phi_1, \dots, \phi_n)$ , 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

$$(\phi_1, \dots, \phi_n) \cdot (\psi_1, \dots, \psi_m) = (\phi_1, \dots, \phi_n, \psi_1, \dots, \psi_m) \quad (\text{concatenate})$$

$$f \circ (\psi_1, \dots, \psi_n) = (f(\psi_1), \dots, f(\psi_n)) \quad (\text{map})$$

$$(\phi_1, \dots, \phi_n) \setminus m = \begin{cases} \epsilon & \text{for } m = 0 \\ (\phi_1, \dots, \phi_m) & \text{for } 0 < m \leq n \\ (\phi_1, \dots, \phi_n) & \text{for } n < m \end{cases} \quad (\text{take})$$

$$(\phi_1, \dots, \phi_n) / m = \begin{cases} (\phi_1, \dots, \phi_n) & \text{for } m = 0 \\ (\phi_{m+1}, \dots, \phi_n) & \text{for } 0 < m \leq n \\ \epsilon & \text{for } n < m \end{cases} \quad (\text{drop})$$

where  $f$  in (map) must be a function whose domain includes the elements of  $\psi$ , and  $m \in \mathbb{N}_0$  (so we can take or drop 0 elements from a sequence). The set of finite sequences over  $\Phi$  is denoted  $\Phi^*$ .

- B. An *infinite sequence* is written  $\phi = (\phi_1, \phi_2, \dots)$  and has  $\#\phi = \infty$  and  $\phi(i) = \phi_i$  for  $1 \leq i$ . The derived sequences are as in the finite case except that

$$(\phi_1, \dots) \cdot (\psi_1, \dots, \psi_m) = (\phi_1, \dots) \quad (\text{concatenate-}\infty)$$

The *identity sequence* is the infinite sequence  $\iota = (1, 2, 3, \dots)$  enumerating all elements of  $\mathbb{N}$ .

- C. A sequence over  $\Phi$  can be used as a *function* (or *map*) from  $\mathbb{N}$  to  $\Phi$  through the convention that  $\phi(i) = \phi_i$  for  $1 \leq i \leq n$ .  $\epsilon$  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  $\phi$  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  $\vec{\phi}$ .

### 2.2.2 Propositions (sequence algebra).

A.  $\phi \cdot (\psi \cdot \gamma) = (\phi \cdot \psi) \cdot \gamma.$

B.  $\epsilon \cdot \phi = \phi \cdot \epsilon = \phi.$

- C.  $\omega \cdot \psi = \omega$  for  $\omega$  infinite.
- D.  $\epsilon \circ \phi = \phi \circ \epsilon = \epsilon$ .
- E.  $\phi \circ \iota = \phi$ ; if  $\phi$  is natural then  $\iota \circ \phi = \phi$ .
- F.  $\phi \circ (\psi \cdot \gamma) = (\phi \circ \psi) \cdot (\phi \circ \gamma)$ .
- G. If  $m + n \leq \#\phi$  then  $\phi/m \setminus n = \phi \setminus (m + n)/m$ .
- H.  $(\iota/n)(m) = n + m$  for  $n, m > 0$ .
- I.  $(\phi \cdot \psi) \setminus m = \begin{cases} \phi \setminus m & \text{if } m < \#\phi \\ \phi \cdot (\psi \setminus (m - \#\phi)) & \text{otherwise} \end{cases}$ .
- J.  $(\phi \cdot \psi)/m = \begin{cases} \phi/m \cdot \psi & \text{if } m < \#\phi \\ \psi/(m - \#\phi) & \text{otherwise} \end{cases}$ .
- K.  $\phi \setminus m = \phi \circ (1, \dots, m)$ .
- L.  $\phi/m = \phi \circ (m + 1, \dots)$ .

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.

**2.2.3 Notation (inductive structures).** When a set is ‘inductively defined by ...’ we use the following metanotation:  $X ::= \dots$  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  $X ::= Y \mid Z$  is a shorthand for the two productions  $X ::= Y$  and  $X ::= Z$ .

The *size* of a term  $t$ , written  $|t|$ , 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  $\epsilon$  instead of nothing to denote ‘empty’.

**2.2.4 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 ‘ $\langle \cdot \cdot \cdot \rangle$ ’ 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  $\{X\}$ .

For each notion of terms it is natural to consider arbitrary subterms and contexts.

**2.2.5 Definitions (contextuality).** Given a notion of T-term defined inductively by some  $t ::= \dots$  productions.

- A. A *T-context* is defined by reusing the same definition but augmenting the production to  $t ::= \dots \mid \square$ , where  $\square$  is a special, unique symbol called *hole*, and considering only terms with exactly one occurrence of  $\square$ .  $C[t]$  is the T-term obtained by inserting  $t$  in place of the hole.
- B. If for some T-terms  $t, t'$  there is a T-context  $C[\ ]$  such that  $t = C[t']$  then  $t'$  is a *subterm* of  $t$ , and conversely,  $t$  is a *superterm* of  $t'$ . The induced *subterm ordering* is written  $t' \trianglerighteq t$ ; it is a partial order.<sup>4</sup>
- C. The *strict subterm ordering* is written  $t' \triangleright t$  and defined by  $t' \triangleright t \wedge t' \neq t$ .
- D. 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  $\square_1, \dots, \square_n$  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):  $\rightarrow$  on an inductively defined set T is contextual if  $t \rightarrow t'$  implies  $C[t] \rightarrow C[t']$  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:

**2.2.6 Definitions (compositionality).** Suppose a notion of T-term is defined inductively by the production

$$t ::= D_1[\vec{t}_1] \mid \dots \mid D_n[\vec{t}_n] \quad (*)$$

where  $D_i[\ ]$  are T-contexts. Consider a predicate  $p$  over  $n$ -tuples of T-terms.

---

<sup>4</sup>The intuition is that it 'points to the root' with the 'subterm at the bottom'.

- A.  $p$  is *flat* if it is defined by a collection of *axioms* of the form

$$p(C[\vec{x}])$$

for T-contexts  $C[]$ .

- B.  $p$  is *compositional* if it is the disjunction (union) of a flat predicate  $p_0$  and a set of *inference rules* of the form

$$\frac{p_1(C_1[\vec{x}], \vec{y}) \wedge \dots \wedge p_n(C_n[\vec{x}], \vec{y})}{p(D_i[\vec{x}], \vec{y})}$$

where the  $C_i[\vec{x}] \triangleright D_i[\vec{x}]$ ,  $i \in \{1, \dots, n\}$ ;  $\vec{x}$  and  $\vec{y}$  are the *inference metavariables* where those in  $\vec{x}$  must be pairwise distinct, and  $C[]$  is any T-context.

- C.  $p$  is *semicompositional* if the same holds except we relax the strict subterm relation  $\triangleright$  to  $\underline{\triangleright}$ , 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 dissertation 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 *derivator* that means a *compositional function*.

**2.2.7 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  $\hat{\triangleleft}$  ‘renaming transformation’ of Launchbury (1993).

Note, however, that any contextual property  $p$  is also compositional because rules such as

$$\frac{p(x_j)}{p(D_i[x_1, \dots, x_n])}$$

can be derived from the contextuality.

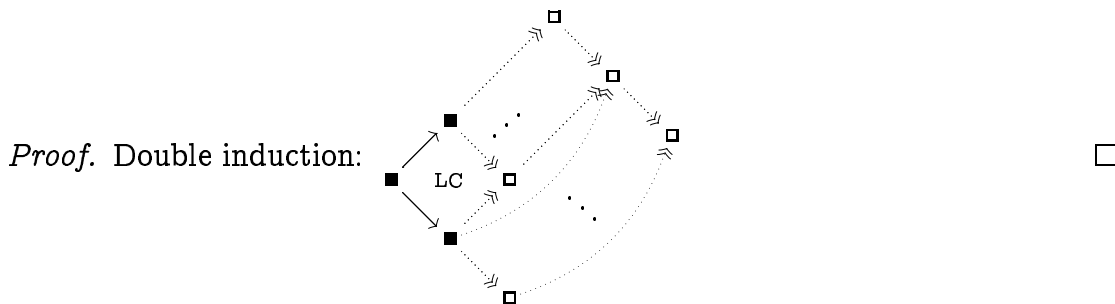
Finally, we present an induction principle for diagrams.



**2.2.8 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:

**2.2.9 Lemma (Newman’s lemma).** If  $\rightarrow$ SN and  $\rightarrow$ LC then  $\rightarrow$ CR (Newman 1942).



## 2.3 $\lambda$ -Calculus

The  $\lambda$ -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  $\lambda$ -calculus.

The  $\lambda$ -calculus was invented by Church (1941); this section summarises the syntax of the  $\lambda\beta$ -calculus following Barendregt (1984) except we use the inductive notations summarised in section 2.2.

**2.3.1 Definition ( $\lambda$ -preterms).** The set of  $\lambda$ -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  $\lambda$ -terms which includes variable binding.

### 2.3.2 Definitions ( $\lambda$ -terms).

- A. The *free variable* set of a  $\lambda$ -term  $M$  is denoted  $\text{fv}(M)$  and defined inductively over  $M$  by  $\text{fv}(x) = \{x\}$ ,  $\text{fv}(\lambda x.M) = \text{fv}(M) \setminus \{x\}$ , and  $\text{fv}(MN) = \text{fv}(M) \cup \text{fv}(N)$ . A  $\lambda$ -preterm  $M$  is *closed* iff  $\text{fv}(M) = \emptyset$ .<sup>5</sup>
- B. 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  $x' \notin \text{fv}(\lambda x.M) \cup \{y, z\}$ , and  $(MN)[y := z] = (M[y := z]) (N[y := z])$ .
- C. Two preterms are  *$\alpha$ -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  $z \notin \text{fv}(MN)$ , and  $MN \equiv PQ$  if  $M \equiv P \wedge N \equiv Q$ .
- D. The set of  *$\lambda$ -terms*,  $\Lambda$ , is the set of  $\lambda$ -preterms modulo  $\equiv$ . The set of *closed  $\lambda$ -terms*,  $\Lambda^\circ$ , is the subset of  $\Lambda$  where the representatives are closed preterms.

The definition of  $\Lambda^\circ$  only makes sense because of the following easily proved property:

**2.3.3 Proposition.** For preterms  $M$  and  $N$ ,  $M \equiv N$  implies  $\text{fv}(M) = \text{fv}(N)$ .

This means that we can safely generalise the notion of free variables and renaming to  $\lambda$ -terms – in fact it means that we can generally confuse preterms and terms which is fortunate because we wish to observe the behaviour of  $\lambda$ -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.

**2.3.4 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.

---

<sup>5</sup>We even allow  $\text{fv}(C[\ ])$  for any context  $C[\ ]$  by the convention that  $\text{fv}(\square) = \emptyset$ .

**2.3.5 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 2.1).

**2.3.6 Definition ( $\beta$ -reduction).** The reduction relation  $\xrightarrow{\beta}$  is the contextual closure of the relation generated by the rule

$$(\lambda x.M)N \rightarrow M[x := N] \quad (\beta)$$

where  $M[x := N]$  is the term obtained by substituting  $N$  for all free occurrences of  $x$  in  $M$ , defined inductively by

$$\begin{aligned} x[y := N] &\equiv \begin{cases} N & \text{if } x = y \\ x & \text{if } x \neq y \end{cases} \\ (\lambda x.M)[y := N] &\equiv \lambda x.M[y := N] \\ (M_1M_2)[y := N] &\equiv M_1[y := N] M_2[y := N] \end{aligned} \quad (*)$$

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  $(\lambda x.x)[x := z] \neq \lambda x.x[x := z]$ .

**Variable capture.** The requirement that bound and free variables are distinct means that  $x \notin \text{fv}(N)$  and thus prevents capturing free variables, *e.g.*, to prevent breaking the distinction  $(\lambda x.y)[y := x] \neq \lambda x.y[y := x]$ .

**2.3.7 Example.** To give the reader an idea of the reduction relation, the full  $\lambda\beta$ -reduction graph of the term  $(\lambda y.(\lambda z.z)y)x$  is  $(\lambda y.(\lambda z.z)y)x \xrightarrow{\beta} (\lambda z.z)x \xrightarrow{\beta} x$ . 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).

**2.3.8 Notation (standard combinators).** A closed  $\lambda$ -term is called a *combinator*, and a growing number of such have been given ‘standard’ names:

$$\begin{aligned} I &\equiv \lambda x.x \quad ; \quad K \equiv \lambda xy.x \quad ; \quad K^* \equiv \lambda xy.y \quad ; \quad S \equiv \lambda xyz.xz(yz) \\ \Omega &\equiv \omega\omega \quad ; \quad \omega \equiv \lambda x.xx \quad (\omega \text{ also called ‘self’}) \\ Y &\equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \quad (Y \text{ is the ‘Curry fixed point combinator’}) \\ \Theta &\equiv AA \quad ; \quad A \equiv \lambda xy.y(xy) \quad (\Theta \text{ is the ‘Turing fixed point combinator’}) \end{aligned}$$

**2.3.9 Remark (extensionality).** The origin of  $\lambda$ -calculus is as a vehicle for expressing the recursive functions, and this is the interest we have in the  $\lambda$ -calculus in this dissertation. However, when studying the relation to *logic* the following additional rewrite rule is important:

$$\lambda x.Mx \rightarrow M \quad \text{if } x \notin \text{fv}(M) \quad (\eta)$$

We will not discuss  $(\eta)$  much in this dissertation.

Finally we reproduce some standard theorems about  $\lambda\beta$ -calculus:

**2.3.10 Proposition (substitution lemma).**

$$M[x := N][y := P] \equiv M[y := P][x := N[y := P]]$$

*Proof.* See Barendregt (1984, 2.1.16). □

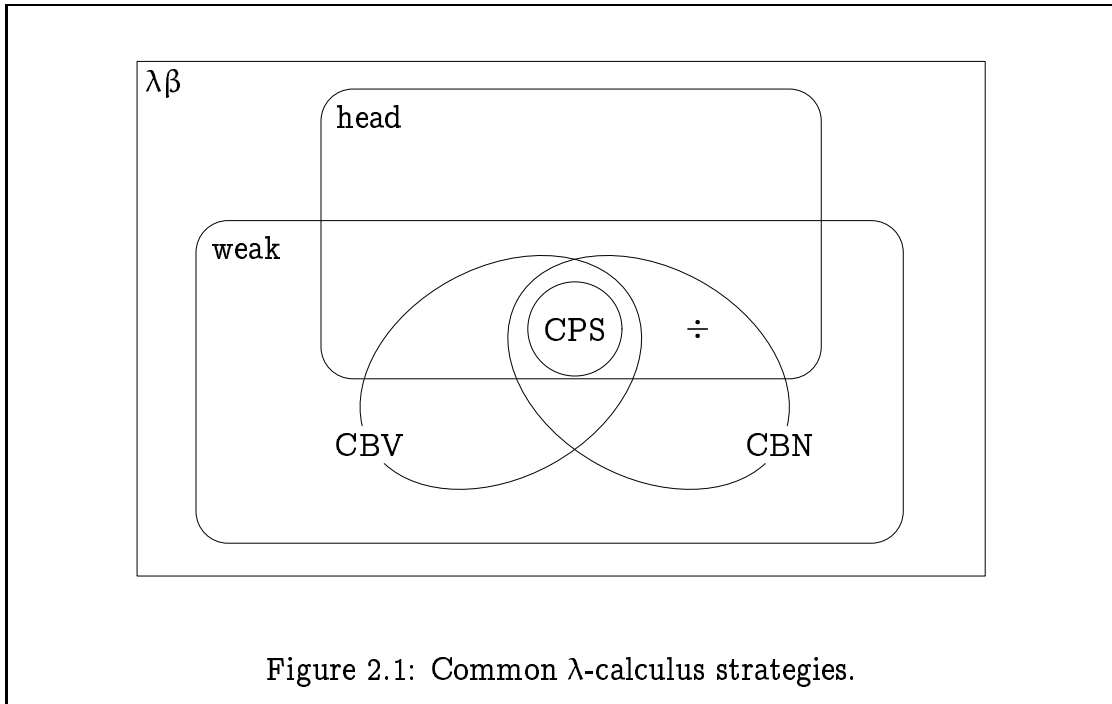
**2.3.11 Theorem (Church-Rosser).**  $\xrightarrow{\beta}$  CR.

*Proof.* See Church and Rosser (1935, Theorem 1). □

## 2.4 Strategies for $\lambda$ -Calculus

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

This section describes the *evaluation strategies* most commonly used in connection with  $\lambda$ -calculus. We will only describe ‘reasonable’ strategies in the



sense that they are all compositional restrictions of the full  $\lambda$ -calculus (this rules out perpetual strategies, for example).

Thus in general strategies are *restrictions* of the  $\xrightarrow{\beta}$  reduction introduced in section 2.3. 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 2.1; 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).

**2.4.1 Definition ( $\wedge$ -inferences).** Contextual  $\lambda$ -calculus can be expressed as a compositional relation by including the following inference rules (rule names from

Curry and Feys 1958):

$$\frac{N \rightarrow N'}{M N \rightarrow M N'} \quad (\mu)$$

$$\frac{M \rightarrow M'}{M N \rightarrow M' N} \quad (\nu)$$

$$\frac{M \rightarrow M'}{\lambda x.M \rightarrow \lambda x.M'} \quad (\xi)$$

Using these rules we can express  $\lambda\beta$ -reduction as the ‘ $\beta\mu\nu\xi$ -calculus’ because it combines the ( $\beta$ ) *axiom* with the ( $\mu$ ), ( $\nu$ ), and ( $\xi$ ) *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  $\beta$ -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).

**2.4.2 History (“ $\beta$ -theories”).** Traditionally inferences are used to define *all* the desired properties of reduction. Such a ‘total’ description is called a *theory*, and the ‘ $\beta$ -conversion theory’ is the ( $\beta$ ) axiom, the ( $\nu$ ), ( $\mu$ ), and ( $\xi$ ) rules, and the following additional *closure rules* that realise the reflexive, symmetric, and transitive properties of  $\beta$ -conversion (rule names from Curry and Feys 1958):

$$M \rightarrow M \quad (\rho)$$

$$\frac{N \rightarrow M}{M \rightarrow N} \quad (\sigma)$$

$$\frac{M \rightarrow N \quad N \rightarrow P}{M \rightarrow P} \quad (\tau)$$

We will not use these rules in this dissertation 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 ‘ $\beta\eta$ -conversion theory’ including ( $\eta$ ) which we will not discuss either.

It turns out that all the strategies that have been invented for evaluation of  $\lambda$ -terms are *weak reduction strategies*: they are restricted to not include  $(\xi)$ . 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.

### 2.4.3 Definitions (whnf).

- A. Weak  $\lambda\beta$ -reduction,  $\xrightarrow{\beta_w}$ , is the relation obtained by omitting  $(\xi)$ .
- B. Head  $\lambda\beta$ -reduction,  $\xrightarrow{\beta_h}$ , is the relation obtained by omitting  $(\mu)$ .

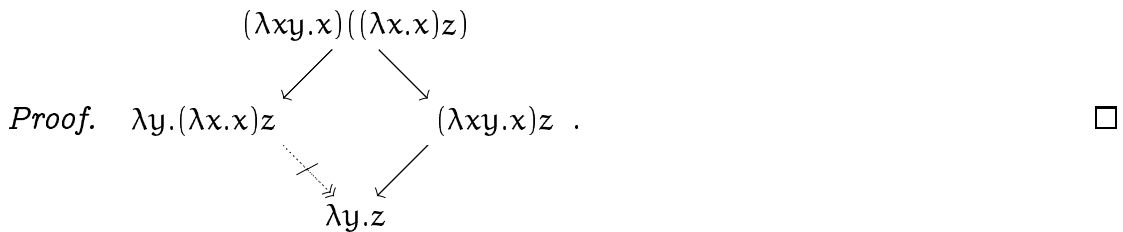
The normal forms of the intersection, omitting  $(\mu)$  and  $(\xi)$ , are said to be in *weak head normal form*, abbreviated whnf, and are easily seen to be of the form

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

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

In general we will be sloppy and call any subreduction of  $\xrightarrow{\beta_w}$  for ‘a weak reduction’ and any subreduction of  $\xrightarrow{\beta_h}$  for ‘a head reduction’. Unfortunately the unrestricted weak reduction is not confluent since redexes can be ‘captured’ underneath a  $\lambda$ :

**2.4.4 Observation.**  $\xrightarrow{\beta_w}$  is not CR.



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.

### 2.4.5 Definitions (Call-by-? strategies).

- A. *Call-by-Value* (CBV) reduction,  $\xrightarrow{V}$ , is defined as the restriction of the  $\lambda\beta$ -calculus to the modified axiom

$$(\lambda x.M)N \rightarrow M[x := N] \quad \text{if } N \text{ is a value} \quad (\beta_V)$$

where “a value” simply means a variable  $x$  or an abstraction  $\lambda x.M$ , and the inferences

$$\frac{N \rightarrow N'}{MN \rightarrow MN'} \quad \text{if } M \text{ is a value} \quad (\mu_V)$$

and  $(\nu)$ .

- B. *Call-by-Name* (CBN) reduction,  $\xrightarrow{N}$ , is defined as the restriction of the  $\lambda\beta$ -calculus to the (original) axiom  $(\beta)$  and the inferences

$$\frac{N \rightarrow N'}{xN \rightarrow xN'} \quad (\mu_N)$$

and  $(\nu)$ .

These are clearly compositional and deterministic (the side condition of  $(\mu_V)$  is not a problem because only two kinds of  $\lambda$ -terms are values, so a simple duplication could eliminate the side condition). They are also weak reductions as neither include  $(\xi)$  in any form. It turns out that CBN implements  $\lambda\beta$  fully whereas CBV is not even normalising:

**2.4.6 Propositions.** A.  $\xrightarrow{V}$  *not* PWN of  $\xrightarrow{\beta}$ . B.  $\xrightarrow{N}$  PWN of  $\xrightarrow{\beta}$ .

*Proofs.* A is easy by considering  $KI\Omega \ll \not\rightarrow \gg I$  but  $KI\Omega \xrightarrow{V} KI\Omega \xrightarrow{V} \dots$ . B is the *standardisation* theorem for closed terms (Plotkin 1975, Theorem 5.1).  $\square$

**2.4.7 History.** Plotkin defines not only the above cited strategies (which are there called ‘left reductions’ because of the inclusion of  $(\nu)$  but also two calculi:

- A. “ $\lambda_V$ ” is  $\lambda\beta$  except  $(\beta)$  is replaced by  $(\beta_V)$ .  
 B. “ $\lambda_N$ ” 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  $\xrightarrow{V}$  and  $\xrightarrow{N}$  (Steele 1978, the CPS name and present formulation origins with).



**2.4.8 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  $\Lambda$  is denoted  $\Lambda_{\text{CPS}}$ .

A more precise view of CPS terms is given by Danvy and Pfenning (1995).

**2.4.9 Propositions.** For all  $M \in \Lambda_{\text{CPS}}$ ,

- A.  $M \xrightarrow{V} N$  implies  $M \xrightarrow{N} N$ , and  $N \in \Lambda_{\text{CPS}}$ , and
- B.  $M \xrightarrow{N} N$  implies  $M \xrightarrow{N} V$ , and  $N \in \Lambda_{\text{CPS}}$ .

*Proof.* Easy case analysis; a side effect is to establish that the area marked with “ $\div$ ” in Figure 2.1 ( $(\text{CBN} \cap \text{head}) \setminus \text{CBV}$ ) is empty.  $\square$

**2.4.10 History.** Abramsky’s (1990) *lazy*<sup>6</sup> calculus,  $\lambda\ell$ , is sometimes claimed as the basis of implementations.  $\lambda\ell$ -reduction is the relation  $\Downarrow_\ell$  inductively defined on closed  $\lambda$ -terms by

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

This is easily seen to be just a ‘big-step’ version of Plotkin’s calculus of Definition 2.4.5.B, restricted to closed terms, *i.e.*,

$$\xrightarrow{N} \Big|_{\Lambda^c} = \Downarrow_\ell$$

(The proof is by isomorphism between  $\xrightarrow{N}$ -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.

---

<sup>6</sup>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 3.5, however, that these definitions are equivalent to usual lazy evaluation when sharing is included.

## 2.5 Namefree $\lambda$ -Calculus

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

The namefree calculus was invented by de Bruijn as a variation of  $\lambda$ -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  $\delta$ -rules. However, we have used a notation which is consistent with the rest of this dissertation.

**2.5.1 Definition ( $\lambda$ NF).** The  $\lambda$ NF-terms are defined inductively by

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

where the letters  $abc$  range over  $\lambda$ NF-terms and  $n$  over natural numbers.  $\beta_{\text{NF}}$ -reduction,  $\xrightarrow{\beta_{\text{NF}}}$  is defined by the relation:

$$(\lambda a)b \rightarrow a[b \cdot \iota] \quad (\beta_{\text{NF}})$$

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 2.2.1 to define substitution;  $\phi$  is always an infinite sequence.

The  $\lambda$ NF-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  $\lambda$ s outside it is 'free', however, its identity is still determined by the 'ghost  $\lambda$ ' to which it belongs. This is reflected by the definition of free variables which is  $\text{fv}(n) = \{n\}$ ,  $\text{fv}(\lambda a) = \{n \mid n + 1 \in \text{fv}(a)\}$ , and  $\text{fv}(ab) = \text{fv}(a) \cup \text{fv}(b)$ .

**2.5.2 Example.** The I, K, and S combinators are written  $\lambda 1$ ,  $\lambda\lambda 2$ , and  $\lambda\lambda\lambda 31(21)$  in  $\lambda$ NF. The reduction of Example 2.3.7 becomes  $(\lambda(\lambda 1)1)1 \xrightarrow[\beta_{\text{NF}}]{\beta_{\text{NF}}} (\lambda 1)1 \xrightarrow{\beta_{\text{NF}}} 1$ . Notice how the outermost 1 is free.

**2.5.3 History.** De Bruijn uses  $S(\phi, a)$  to denote the substitution metanotion (our  $a[\phi]$  with reversed  $\phi$ ), and  $\tau_h$  for  $\iota/h$ .

It is possible to relate  $\Lambda$  and  $\Lambda\text{NF}$ , of course: the following inference system does this, assuming the list of free variables is  $\rho$  (corresponding to the sequence of ‘ghost  $\lambda$ s’):

**2.5.4 Definition (translation,  $\lambda\text{NF}/\lambda$ ).** The closed  $\lambda\text{NF}$ -term  $a$  projects onto the  $\lambda$ -term  $M$ , notation  $a \triangleright M$ , if we can prove  $\epsilon \vdash a \triangleright M$  in the following system:

$$\begin{array}{c} \rho \vdash n \triangleright x \quad \rho(n) = x \\ \\ \frac{x \cdot \rho \vdash a \triangleright M}{\rho \vdash \lambda a \triangleright \lambda x.M} \quad x \notin \rho \\ \\ \frac{\rho \vdash a \triangleright M \quad \rho \vdash b \triangleright N}{\rho \vdash ab \triangleright MN} \end{array}$$

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

## 2.6 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  $\lambda$ -calculus, by formalizing precisely the intuitive notion of substitution without extra constraints such as typing, algebraic conventions, *etc.* We study CRSs in this dissertation because they constitute a generalisation of the rewriting done by functional programming ‘evaluation steps.’ An *implementation* of CRS reduction is provided by chapter 6.

The CRS formalism was invented by Klop (1980) to provide a systematic treatment of combinations of term rewrite systems (TRS) with the  $\lambda$ -calculus, inspired by the ‘definable extensions of the  $\lambda$ -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 chapter 6).

### 2.6.1 Definitions (combinatory reduction systems).

- A. An *alphabet* is a set of *function symbols*  $A, B, C, F, G, \dots$  (and sometimes Greek letters and exotic symbols); each annotated with a *fixed arity* (notation  $F^n$ ).
- B. The corresponding *preterms*, denoted  $t, s, \dots$ , have the inductive form

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

where  $x, y, z, \dots$  are used to denote *variables* and  $X, Y, Z$  to denote *meta-variables*, each with an explicit arity superscript as indicated.<sup>7</sup> The four different preterm forms are called *variable*, *metaabstraction*, *construction*, and *metaapplication* of a metavariable, respectively.

- C. We denote the *set of metavariables* occurring in a preterm  $t$  as  $mv(t)$ .
- D. A preterm is *closed* if  $fv(t) = \emptyset$  where  $fv(x) = \{x\}$ ,  $fv([x]t) = fv(t) \setminus \{x\}$ , and  $fv(F^n(t_1, \dots, t_n)) = fv(Z^n(t_1, \dots, t_n)) = fv(t_1) \cup \dots \cup fv(t_n)$ .
- E. Metaterms are closed preterms considered *modulo renaming of (bound) variables*: we take as implicit the usual  $\alpha$ -equivalence denoted  $\equiv$  and defined inductively by  $x \equiv x$ ,  $[x]t \equiv [y]s$  if  $t[x := z] \equiv s[y := z]$  for  $z \notin fv(s) \cup fv(t)$ , 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  $x' \notin fv([x]t) \cup \{y, z\}$ ,  $F^n(t_1, \dots, t_n)[y := z] = F^n(t_1[y := z], \dots, t_n[y := z])$ , and  $Z^n(t_1, \dots, t_n)[y := z] = Z^n(t_1[y := z], \dots, t_n[y := z])$ .
- F. A *terms* is a metaterm without metavariables (but possibly including metaabstraction which is then called *abstraction*).
- G. *Rewrite rules*, written  $p \rightarrow r$ , 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.

---

<sup>7</sup>Thus for each particular alphabet the definition of preterms is finite and inductive.

- the RHS (right hand side)  $r$  must be a *contractor* of the LHS by which we mean that  $mv(p) \supseteq mv(r)$ .

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

**2.6.2 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.

**2.6.3 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).

**2.6.4 Example ( $\lambda$ -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

$$@^2(\lambda^1([x]Z^1(x)), Y^0()) \rightarrow Z^1(Y^0()) \quad (\beta)$$

One can verify easily that  $(\beta)$  will reduce a restricted term to another restricted term.

The term  $Z^1(Y^0())$  on the righthand side of  $(\beta)$  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  $\lambda$ -term  $(\lambda x.yx)y$  corresponds to the CRS term  $@(\lambda([x]@(y,x)),y)$  and reduces by  $(\beta)$  to  $@(y,y)$  if  $Z^1(x)$  is matched with  $@(y,x)$ .

**2.6.5 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  $[x][y][z]t$ ,
- $F_{xyz}.t$  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  $x_1, \dots, x_n$ ,  $Z_1, \dots, Z_n$ , and  $t_1, \dots, t_n$ , 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  $\lambda$ -calculus to be followed, e.g.,  $\lambda xyz.xz(yz)$  denotes the rather unwieldy  $\lambda^1([x](\lambda^1([y](\lambda^1([z]@^2(@^2(x,z),@^2(y,z))))))$ .

**2.6.6 Example ( $\lambda$ -calculus, readable CRS).** Using the above abbreviations we can express the  $\lambda\beta$ -calculus in a more usual notation: the following CRS is exactly the same as the one in Example 2.6.4 above:

$$(\lambda x.Z(x))Y \rightarrow Z(Y) \quad (\beta)$$

**2.6.7 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:

$$SXYZ \rightarrow (XZ)(YZ) \quad (S)$$

$$KXY \rightarrow X \quad (K)$$

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

The substitution concept of CRS is reminiscent of a two-level  $\lambda$ -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.

**2.6.8 Definition (substitution).** A *valuation*  $\sigma$  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  $t[x_1 := t_1, \dots, x_n := t_n]$  defined inductively by

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

(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  $\sigma$  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.

### 2.6.9 Definitions (safeness).

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

- B. The rewrite rule  $p \rightarrow t$  is *safe for the valuation*  $\sigma$  if for  $p$  and  $t$  considered as preterms

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

- C. The valuation  $\sigma$  is *safe with respect to itself* if

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

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

**2.6.10 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.

**2.6.11 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:

- *TRSs*, 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 1996).

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

- A.  $C[\sigma(p)]$  is *reducible*,  $\sigma(p)$  is the *redex*, and  $C[\ ]$  is its *occurrence*.



- B.  $C[\sigma(r)]$  is the *contractum* of the rule, and
- C. the pair  $C[\sigma(p)] \rightarrow C[\sigma(r)]$  is a *rewrite* of the rule.

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

**2.6.13 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 dissertation 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):

**2.6.14 Definitions (orthogonality).** Let  $R$  be a CRS which consist of rules  $p_i \rightarrow \dots$

- A.  $R$  is *left-linear* if there are no repeated metavariables in any of the  $p_i$ .
- B. All  $p_i$  must have the form  $C[Z_1^{a_1}(\vec{x}_{1(a_1)}), \dots, Z_n^{a_n}(\vec{x}_{n(a_n)})]$  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.
- C.  $R$  is *orthogonal* if it is left-linear and nonoverlapping (some authors call this *regular*).

**2.6.15 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*,  $\multimap$ , 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*<sup>8</sup> 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 dissertation.

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

**2.6.16 Example (symbolic differentiation).** Let the *arithmetic functions* (unary), ranged over by  $F$ , and the *arithmetic expressions*, ranged over by  $A$  and  $B$ , be defined inductively by

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

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 2.2 (based on the list in Spiegel (1968, section 13)). One can use this to differentiate simple functions such as  $\ln(x + 1)$ :

$$\begin{aligned} &\boxed{d(\lambda x. (\ln)(x + 1))} \xrightarrow{\text{(function)}} \lambda x. \boxed{D([\!x\!](\ln)(x + 1), x)} \xrightarrow{\text{(chain)}} \\ &\lambda x. \left( \boxed{d(\ln)}_{(\ln)} \right) (x + 1) \times \boxed{D([\!x\!]x + 1, x)}_{(\text{plus})} \\ &\rightsquigarrow \lambda x. \boxed{\left( \lambda x. \frac{1}{x} \right) (x + 1)}_{(\beta)} \times \left( \boxed{D([\!x\!]x, x)}_{(\text{identity})} + \boxed{D([\!x\!]1, x)}_{(\text{constant})} \right) \\ &\rightsquigarrow \lambda x. \frac{1}{x + 1} \times \boxed{(1 + 0)}_{(x+0)} \xrightarrow{\text{(x1)}} \lambda x. \boxed{\frac{1}{x + 1} \times 1}_{(x1)} \xrightarrow{\text{(x1)}} \lambda x. \frac{1}{x + 1} \end{aligned}$$

(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  $E$ ).

---

<sup>8</sup>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  $f$ .

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

Next the differentials of some standard functions.

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

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).

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

$$D([x]x, Z) \rightarrow 1 \quad (\text{identity})$$

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

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

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

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

The interaction between functions is described as follows.

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

Finally applications need to be contracted.

$$(\lambda x.X(x))Y \rightarrow X(Y) \quad (\beta)$$

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

$$0 + X \rightarrow X \quad (0+x) ; \quad X + 0 \rightarrow X \quad (x+0) ; \quad X - 0 \rightarrow X \quad (x-0)$$

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

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

Figure 2.2: Symbolic differentiation CRS.

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

$$\begin{aligned}
& \boxed{d(\lambda z.(\lambda x.(\ln)(x+1))((\lambda x.\ln x)z))} \\
& \quad \text{(function)} \\
& \longrightarrow \lambda x. \boxed{D([x](\lambda b.(\ln)(b+1))((\lambda c.\ln c)x), x)} \\
& \quad \text{(chain)} \\
& \longrightarrow \lambda x. \left( \boxed{d(\lambda a.(\ln)(a+1))} \right) \left( \boxed{(\lambda e.\ln e)x} \right) \times \boxed{(D([x](\lambda e.\ln e)x, x))} \\
& \quad \text{(function)} \quad \quad \quad \text{(\beta)} \quad \quad \quad \text{(chain)} \\
& \rightsquigarrow \lambda x. \boxed{(\lambda x.D([x](\ln)(x+1), x))(\ln x)} \times \left( \left( \boxed{d(\lambda a.\ln a)} \right) x \times \boxed{(D([x]x, x))} \right) \\
& \quad \quad \quad \text{(\beta)} \quad \quad \quad \text{(function)} \quad \quad \quad \text{(identity)} \\
& \rightsquigarrow \lambda x. \boxed{(D([b](\ln)(b+1), \ln x))} \times \boxed{((\lambda x.D([x]\ln x, x))x \times 1)} \\
& \quad \quad \quad \text{(chain)} \quad \quad \quad \text{(x1)} \\
& \rightsquigarrow \lambda x. \left( \left( \boxed{d(\ln)} \right) (\ln x + 1) \times \boxed{(D([x]x + 1, \ln x))} \right) \times \boxed{(\lambda a.D([b]\ln b, a))x} \\
& \quad \quad \quad \text{(ln)} \quad \quad \quad \text{(plus)} \quad \quad \quad \text{(\beta)} \\
& \rightsquigarrow \lambda x. \left( \left( \boxed{\lambda x. \frac{1}{x}} (\ln x + 1) \right) \times \left( \boxed{(D([x]x, \ln x))} + \boxed{(D([x]1, \ln x))} \right) \right) \times \boxed{(D([d]\ln d, x))} \\
& \quad \quad \quad \text{(\beta)} \quad \quad \quad \text{(identity)} \quad \quad \quad \text{(constant)} \quad \quad \quad \text{(chain)} \\
& \rightsquigarrow \lambda x. \left( \frac{1}{\ln x + 1} \times \boxed{(1+0)} \right) \times \left( \left( \boxed{d(\ln)} \right) x \times \boxed{(D([x]x, x))} \right) \\
& \quad \quad \quad \text{(x+0)} \quad \quad \quad \text{(ln)} \quad \quad \quad \text{(identity)} \\
& \rightsquigarrow \lambda x. \left( \frac{1}{\ln x + 1} \times 1 \right) \times \left( \left( \lambda x. \frac{1}{x} \right) x \times 1 \right) \rightsquigarrow \lambda x. \frac{1}{\ln x + 1} \times \left( \lambda a. \frac{1}{a} \right) x \\
& \quad \quad \quad \text{(x1)} \quad \quad \quad \text{(x1)} \quad \quad \quad \text{(\beta)} \\
& \longrightarrow \lambda x. \frac{1}{\ln x + 1} \times \frac{1}{x} \longrightarrow \lambda x. \frac{\boxed{1 \times 1}}{(\ln x + 1) \times x} \longrightarrow \lambda x. \frac{1}{(\ln x + 1) \times x} \\
& \quad \quad \quad \text{(1/x)} \quad \quad \quad \text{(1x)}
\end{aligned}$$

## 2.7 Summary.

We have presented a host of notations and standard results: everything in this chapter has been published in the literature. The presentation is original, however, in particular the exposition of *translation* and *projection* in Definition 2.1.10, of *compositionality* in Definition 2.2.6, and of *reduction strategies* in section 2.4.

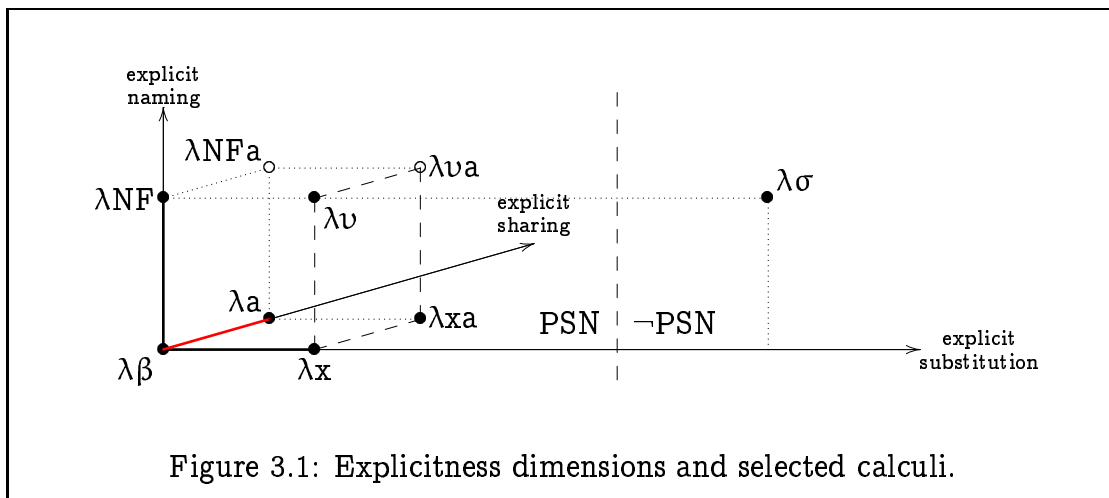
# 3

## Explicit Conservative Extensions of $\lambda$ -calculus

This chapter is devoted to *conservative extensions* of the  $\lambda\beta$ -calculus: this is the chapter devoted to the study of making explicit those aspects of mechanical evaluation that neither increase nor reduce the expressive power. In particular all the discussed calculi will be confluent and contextual.

We will study *syntactic extensions* in that each extension makes some operational aspect of reduction explicit. Specifically, we show how extensions giving explicit sharing, explicit naming, 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  $\lambda$ -calculi can be understood as ‘points in  $\lambda$ -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.

The situation is illustrated in Figure 3.1: the origin is the (untyped)  $\lambda\beta$ -calculus, and we give samples of a ‘unit’ of each dimension (where the dimension is first fully explicit):  $\lambda\text{NF}$  is de Bruijn’s (1972) ‘namefree’ calculus described in section 2.5; we will say that this has *explicit naming*.  $\lambda_x$  is our calculus with *explicit substitution* in the most abstract way. Finally,  $\lambda_a$  is a variation of Wadsworth’s (1971)  $\lambda$ -graph calculus using (abstract) *addresses* which we will say has *explicit sharing*.



We proceed as follows: First, in section 3.1, we present  $\lambda_{xgc}$ , a calculus of *explicit substitution* which retains ‘implicit’ variable names which is the  $\lambda_x$  discussed above with a technical extension, namely a rule for *explicit garbage collection* which will turn out to play an important rôle in many proofs even though it adds nothing to the expressiveness of the calculus. In particular we use it essentially in the following section 3.2 to give a direct proof that  $\lambda_{xgc}$  preserves strong normalisation of  $\lambda\beta$ -reduction.

Second, we investigate, in section 3.3, the interaction between *explicit naming* and explicit substitution: first we compare the  $\lambda_x$ -calculus with  $\lambda_u$  of Lescanne (1994a), and obtain a projection from  $\lambda_u$  into  $\lambda_x$  which we show to maintain the preservation of  $\lambda\beta$  strong normalisation, thus obtaining a constructive proof for PSN of  $\lambda\beta$  for  $\lambda_u$ . We then generalise the properties of this projection to a notion of *strict projection* which we use to generalise the proof to all the calculi for which PSN of  $\lambda\beta$  is already known. This teaches us that the explicit substitution dimension is the critical one for PSN of  $\lambda\beta$ , and we discuss the ‘boundary’ shown as a dashed line on Figure 3.1 in this direction, where  $\lambda\sigma$  is known to be on the ‘wrong side’, confirming the result reported by Melliès (1995).

Third, we study the *explicit sharing* dimension in section 3.4: we introduce sharing in general and for  $\lambda$ -calculus in particular, a major part of the latter is a comparison to Wadsworth’s (1971)  $\lambda$ -graph reduction.

In section 3.5 we synthesise explicit sharing and substitution into  $\lambda_{xa}$  with which we can model the *complexity of sharing* independently of the reduction strategy: This gives the first calculus in this dissertation where the *reduction*

*count* is a realistic complexity measure and the term *size truly* measures the space consumption.

Parts of the chapter have been published/presented (Rose 1995, Bloo and Rose 1995, Rose and Bloo 1995).

## 3.1 Explicit Substitution

This section presents the  $\lambda xgc$ -calculus: an *explicit substitution calculus* in the tradition of  $\lambda\sigma$  of Abadi, Cardelli, Curien and Lévy (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,  $\lambda xgc$  shares with  $\lambda s$  of Kamareddine and Ríos (1995) and  $\lambda v$  of Lescanne (1994a) that there is no syntax for explicit composition of substitutions (we elaborate on this in section 3.2). 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  $\lambda x$ -terminology which we will use throughout the chapter, and generalise some standard related concepts from the  $\lambda$ -calculus of section 2.3 to them. Then we present  $\lambda xgc$ -reduction. Finally we analyse properties of the substitution and garbage collection subrelations and explain the relation to standard substitution and  $\beta$ -reduction, showing that the reduction relation of  $\lambda xgc$  is a conservative extension of  $\xrightarrow{\beta}$ , from which confluence of  $\lambda xgc$  follows.

**3.1.1 Definition ( $\lambda x$ -preterms).** The  $\lambda x$ -preterms is the extension of the  $\lambda$ -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  $x$ s in all of the subterms  $M$ ,  $N$ ,  $P$ , and  $Q$ , are bound by the  $\lambda$ ).

The  $\lambda x$ -terms include as a subset the ordinary  $\lambda$ -terms,  $\Lambda$ ; we will say that a  $\lambda x$ -term is ‘pure’ if it is also a  $\lambda$ -term, *i.e.*, if it has no subterms of the form

$M\langle x := N \rangle$ . The usual  $\beta$ -reduction is denoted  $\xrightarrow{\beta}$  (as defined on the pure terms only, of course, cf. Definition 2.3.6). All the usual standard concepts generalise naturally.

### 3.1.2 Definitions ( $\lambda x$ -terms).

- A. The *free variable* set of a  $\lambda x$ -preterm  $M$  is denoted  $\text{fv}(M)$  and defined inductively over  $M$  by (the usual)  $\text{fv}(x) = \{x\}$ ,  $\text{fv}(\lambda x.M) = \text{fv}(M) \setminus \{x\}$ ,  $\text{fv}(MN) = \text{fv}(M) \cup \text{fv}(N)$ , and (the new)

$$\text{fv}(M\langle x := N \rangle) = (\text{fv}(M) \setminus \{x\}) \cup \text{fv}(N) .$$

A  $\lambda x$ -preterm  $M$  is *closed* iff  $\text{fv}(M) = \emptyset$ .

- B. 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  $x' \notin \text{fv}(\lambda x.M) \cup \{y, z\}$ ,  $(MN)[y := z] = (M[y := z]) (N[y := z])$ , and (the new)

$$(M\langle x := N \rangle)[y := z] = M[x := x'] [y := z] \langle x' := N[y := z] \rangle$$

with  $x' \notin \text{fv}(\lambda x.M) \cup \{y, z\}$ .

- C. That two terms are  *$\alpha$ -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  $z \notin \text{fv}(MN)$ ,  $MN \equiv PQ$  if  $M \equiv P$  and  $N \equiv Q$ , and (the new)

$$M\langle x := N \rangle \equiv P\langle y := Q \rangle \text{ if } N \equiv Q \text{ and } M[x := z] \equiv P[y := z] \text{ for } z \notin \text{fv}(MP).$$

- D. The set of  $\lambda x$ -terms,  $\Lambda x$ , is the set of  $\lambda x$ -preterms modulo  $\equiv$ . The set of *closed  $\lambda x$ -terms*,  $\Lambda x^\circ$ , is the subset of  $\Lambda$  where the representatives are closed preterms.

**3.1.3 Remark.** One might argue that

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

is a better definition since when  $x \notin \text{fv}(M)$  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 2.3.4.

**3.1.4 Definitions ( $\lambda xgc$ -reduction).** Define the following reductions on  $\lambda x$ .

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

$$(\lambda x.M)N \rightarrow M\langle x := N \rangle \quad (b)$$

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

$$\begin{aligned} x\langle x := N \rangle &\rightarrow N && (xv) \\ x\langle y := N \rangle &\rightarrow x \quad \text{if } x \neq y && (xvgc) \\ (\lambda x.M)\langle y := N \rangle &\rightarrow \lambda x.M\langle y := N \rangle && (xab) \\ (M_1M_2)\langle y := N \rangle &\rightarrow M_1\langle y := N \rangle M_2\langle y := N \rangle && (xap) \end{aligned}$$

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

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

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

D.  $\lambda xgc$ -reduction is  $\xrightarrow{bxgc} = \xrightarrow{b} \cup \xrightarrow{x} \cup \xrightarrow{gc}$ ,  $\lambda x$ -reduction is  $\xrightarrow{bx} = \xrightarrow{b} \cup \xrightarrow{x}$ , and  $\xrightarrow{xgc} = \xrightarrow{x} \cup \xrightarrow{gc}$ .

The following is immediate from the definition.

**3.1.5 Propositions.** A. If  $M, N \in \Lambda$ ,  $M\langle x := N \rangle \xrightarrow{x} M[x := N]$ . B.  $P \xrightarrow{x} Q$  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.

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

**3.1.7 Remark (implicit renaming).** The above reduction definition is based directly on the usual definition of  $\beta$ -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 2.3.4. 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  $x \neq y$ ) and it avoids capture by requiring that  $x \notin \text{fv}(N)$ . Alternatively, this could be achieved by *explicit renaming*, *i.e.*, by using the following equivalent but more cumbersome formulation of the rule:

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

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  $\lambda x \dots$  would capture them. This is prevented by renaming  $x$  in  $M$  to  $x'$ , requiring that  $x' \notin \text{fv}(N)$ .
- 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  $\lambda x' \dots$ . This is ensured by the constraint  $x' \notin \text{fv}(\lambda x.M)$ .
- Finally, the new variable should not clash with  $y$ , hence the requirement  $x' \notin \{y\}$ .

Notice that if  $x \notin \text{fv}(N)$  and  $x \neq y$ , there is no need to rename:  $x \equiv x'$  is acceptable.

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

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

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

**3.1.8 Example.** To give the reader an idea of this new reduction relation we present the full  $\lambda xgc$ -reduction graph of the term  $(\lambda y.(\lambda z.z)y)x$  in Figure 3.2 (the pure terms are boxed). For comparison the full  $\beta$ -reduction graph of this term is

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

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  $(\lambda z.z)x$ .

**3.1.9 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  $\lambda$ -conversion” of Revesz (1985):

$$(\lambda x.x)Q \rightarrow Q \tag{\beta 1}$$

$$(\lambda x.x)y \rightarrow x \quad \text{if } x \neq y \tag{\beta 2}$$

$$(\lambda x.\lambda x.P)Q \rightarrow \lambda x.P \tag{\beta 3}$$

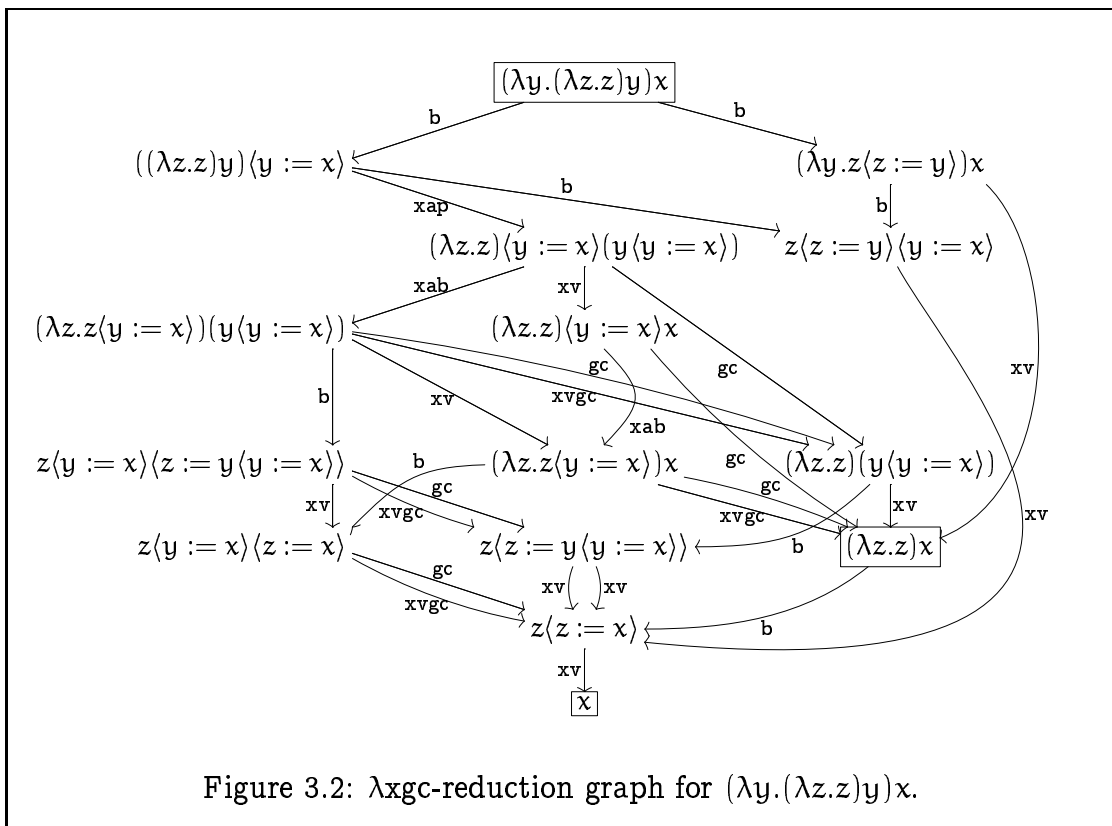
$$(\lambda x.\lambda y.P)Q \rightarrow \lambda y.(\lambda x.P)Q \quad \text{if } x \neq y \text{ and } x \notin \text{fv}(P) \text{ and } x \notin \text{fv}(Q) \tag{\beta 4}$$

$$(\lambda x.P_1P_2)Q \rightarrow (\lambda x.P_1)Q ((\lambda x.P_2)Q) \tag{\beta 5}$$

This relation, call it  $\overline{\mathcal{R}}$ , is related to  $\lambda x$  by the equation  $\overline{\mathcal{R}} \rightarrow = \overline{\mathcal{B}} \cdot \overline{\mathcal{X}} \cdot \overline{\mathcal{B}}^\dagger$  (the last arrow is  $\mathcal{B}$ -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 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:

$$(\lambda x.a)b \rightarrow a[(b/x) \cdot \text{id}] \tag{Beta}$$



$$x[(a/x) \cdot s] \rightarrow a \quad (\text{Var1})$$

$$x[(a/y) \cdot s] \rightarrow x[s] \quad (\text{Var2})$$

$$x[id] \rightarrow x \quad (\text{Var3})$$

$$(ba)[s] \rightarrow (b[s])(a[s]) \quad (\text{App})$$

$$(\lambda x.a)[s] \rightarrow \lambda y.(a[(y/x) \cdot s]) \quad \text{if } y \text{ does not occur in } a \quad (\text{Abs})$$

The rules correpond closely to those of  $\lambda x$  except for the crucial difference that (Abs) introduces *explicit renaming* as discussed in Remark 3.1.7 – the ‘names’ are *not* variable names in the  $\lambda$ -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  $\lambda$ -calculus, in particular when explaining the relationship between explicit and traditional (implicit) substitution, and garbage collection.

**3.1.10 Propositions.** A.  $\xrightarrow{b} \text{SN}$ , B.  $\xrightarrow{b} \diamond$ , C.  $\xrightarrow{xgc} \text{SN}$ , D.  $\xrightarrow{xgc} \text{CR}$ , and E.  $\xrightarrow{xgc} \text{UN}$ .

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

$\xrightarrow{xgc} \text{SN}$  is shown by finding a map  $h : \Lambda x \rightarrow \mathbb{N}$  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  $h(M\langle x := N \rangle) = h(M) \times (h(N) + 1)$ .

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

$$x\langle y := N \rangle \begin{array}{c} \xrightarrow{xvgc} \\ \xrightarrow{gc} \end{array} x$$

where  $y \neq x$ ,

$$\begin{array}{ccc} & \lambda x.M\langle y := N \rangle & \\ & \nearrow \text{xab} & \searrow \text{gc} \\ (\lambda x.M)\langle y := N \rangle & \xrightarrow{\text{gc}} & \lambda x.M \end{array}$$

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

$$\begin{array}{ccc} M_1\langle y := N \rangle & M_2\langle y := N \rangle & \\ & \nearrow \text{xap} & \searrow \text{gc} \\ (M_1M_2)\langle y := N \rangle & \xrightarrow{\text{gc}} & (M_1M_2) \end{array}$$

where  $y \notin \text{fv}(M_1M_2)$ . SN and LC imply CR and UN by Lemma 2.2.9.  $\square$

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  $P[x := Q]$ .

**3.1.11 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  $M, N_1, \dots, N_n$  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  $x \neq x_1$  then

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

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

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

□

An interesting consequence of representation is the following:

**3.1.12 Corollary (substitution lemma).**

$$M\langle x := N \rangle\langle y := P \rangle \xleftarrow{\text{xgc}} M\langle y := P \rangle\langle x := N\langle y := P \rangle \rangle$$

*Proof.* Follows from Lemma 3.1.11 by the  $\lambda$ -calculus substitution lemma. □

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

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

**3.1.13 Propositions.** For any  $M$ ,

- A. If  $M \xrightarrow{\text{gc}} N$  then  $\text{fv}(M) \supseteq \text{fv}(N)$  and  $\downarrow_x(M) \equiv \downarrow_x(N)$ .
- B. If  $M \xrightarrow{x} N$  then  $\text{fv}(M) \supseteq \text{fv}(N)$ .
- C. If  $M \xrightarrow{\text{b}} N$  then  $\text{fv}(M) = \text{fv}(N)$ .
- D. If  $M$  is garbage-free then  $\text{fv}(M) = \text{fv}(\downarrow_x(M))$ .

*Proofs.* The two free variable inclusions and c 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  $P\langle x := Q \rangle \xrightarrow{\text{gc}} P$  where  $x \notin \text{fv}(P)$ : then by Lemma 3.1.11  $\downarrow_x(P\langle x := Q \rangle) \equiv \downarrow_x(P)[x := \downarrow_x(Q)] \equiv \downarrow_x(P)$  since  $x \notin \text{fv}(P) \supseteq \text{fv}(\downarrow_x(P))$ .

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

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

where  $M_{i+1} \equiv M'_{i+1}$  if  $M'_i \xrightarrow{x} M_{i+1}$  is not an  $(xab)$  step, and  $M_{i+1} \xrightarrow{gc} M'_{i+1}$  if  $M'_i \xrightarrow{x} M_{i+1}$  is a  $(xab)$  step, and all  $M'_i$  are garbage-free. For instance if  $M'_i \equiv (PQ)\langle x := R \rangle$ ,  $M_{i+1} \equiv (P\langle x := R \rangle)(Q\langle x := R \rangle)$ ,  $x \in \text{fv}(P)$ ,  $x \notin \text{fv}(Q)$ , 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  $\text{fv}(M'_i) = \text{fv}(M'_{i+1})$ .  $\square$

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

A consequence of Proposition 3.1.13.A is that we do not need to introduce a special notation for  $\xrightarrow{gc}$ -normal forms since any pure term is also garbage-free, *i.e.*,  $\xrightarrow{gc} \equiv \xrightarrow{x}$ .

**3.1.14 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  $\xrightarrow{x} \not\subseteq \xrightarrow{gc}$  but also  $\xrightarrow{gc} \not\subseteq \xrightarrow{x}$ : the second inclusion fails because a single  $\xrightarrow{gc}$  can remove a single substitution which can't be removed by  $\xrightarrow{x}$  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  $\xrightarrow{x}$  (also see Remark 3.1.3). This is witnessed in Figure 3.2 where most but not all  $\xrightarrow{gc}$ -arrows are adjacent to an  $\xrightarrow{xvgc}$ -arrow.

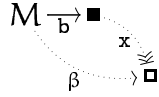
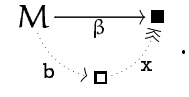
Furthermore,  $\xrightarrow{gc}$  permits more efficient reduction through the use of clever strategies. One of the problems with explicit substitution is that there is a tendency to create substitutions that do not bind a variable, as is the case in  $(x\langle fyyy \rangle)\langle x := I \rangle \xrightarrow{x} x\langle x := I \rangle\langle fyyy \rangle\langle x := I \rangle$ . 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)  $\beta$ -reduction. We show that  $\xrightarrow{bxgc}$  is a

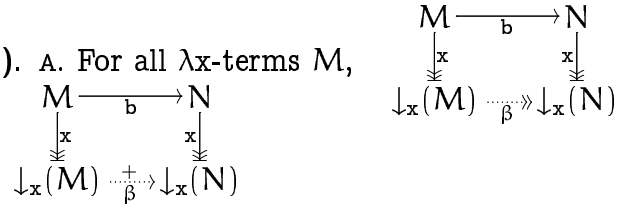
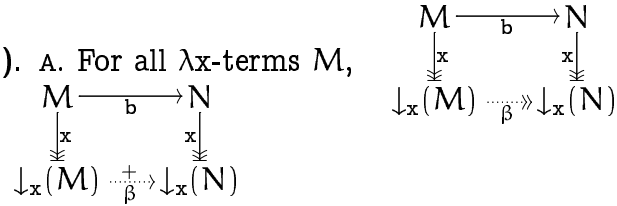


conservative extension of  $\xrightarrow{\beta}$ , and thus confluent. We first relate single reduction steps and finally full reduction.

**3.1.15 Proposition.** For pure  $M$ , A. , and B. .

*Proof.* Both follow from the observation that we can choose to contract the same  $\beta$ -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 3.1.11.  $\square$

Next Proposition 3.1.15.A is generalised to non-pure terms because we will need this in the confluence proof below. In fact we also strengthen 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.

**3.1.16 Lemmas (projection).** A. For all  $\lambda x$ -terms  $M$ ,  B. For garbage-free  $M$ , 

*Proof.* We first prove B by induction on the structure of  $M$ :

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

$$\downarrow_x(M) \equiv (\lambda x. \downarrow_x(P)) \downarrow_x(Q) \xrightarrow{\beta} \downarrow_x(P)[x := \downarrow_x(Q)] \stackrel{\text{Lemma 3.1.11}}{\equiv} \downarrow_x(P\langle x := Q \rangle)$$

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

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  $\downarrow_x(P) \xrightarrow{\beta} \downarrow_x(P')$ , hence also  $\downarrow_x(P)[x := \downarrow_x(Q)] \xrightarrow{\beta} \downarrow_x(P')[x := \downarrow_x(Q)]$ . Thus  $\downarrow_x(M) \stackrel{\text{Lemma 3.1.11}}{\equiv} \downarrow_x(P)[x := \downarrow_x(Q)] \xrightarrow{\beta} \downarrow_x(P')[x := \downarrow_x(Q)] \stackrel{\text{Lemma 3.1.11}}{\equiv} \downarrow_x(N)$ .

**Case**  $M \equiv P\langle x := Q \rangle$ ,  $N \equiv P\langle x := Q' \rangle$ : By IH we know that  $\downarrow_x(Q) \xrightarrow{+}_{\beta} \downarrow_x(Q')$ , hence also  $\downarrow_x(P)[x := \downarrow_x(Q)] \xrightarrow{+}_{\beta} \downarrow_x(P)[x := \downarrow_x(Q')]$  since by the fact that  $M$  is garbage-free and Proposition 3.1.13.D,  $x \in \text{fv}(\downarrow_x(P))$ . Thus  $\downarrow_x(M) \stackrel{\text{Lemma 3.1.11}}{\equiv} \downarrow_x(P)[x := \downarrow_x(Q)] \xrightarrow{+}_{\beta} \downarrow_x(P)[x := \downarrow_x(Q')] \stackrel{\text{Lemma 3.1.11}}{\equiv} N$ .

$A$  is analogous except that we do not know in the last case that  $x \in \text{fv}(\downarrow_x(P))$  and hence have to use  $\xrightarrow{-}_{\beta}$ , allowing identity ( $\equiv$ ).  $\square$

The above suffices to show that  $\lambda xgc$  is indeed a conservative extension of the  $\lambda\beta$ -calculus, with  $\xrightarrow{-}_x$  as translation.

**3.1.17 Theorem.** *For pure terms  $M, N$ :*  $M \xrightarrow{-}_{bxgc} N \iff M \xrightarrow{-}_{\beta} N$

*Proof.* **Case**  $\Leftarrow$ : Assume  $M \xrightarrow{-}_{\beta} N$ ; the case then follows by induction on the length of the  $\xrightarrow{-}_{\beta}$ -reduction, using Proposition 3.1.15.B in each step.

**Case**  $\Rightarrow$ : Assume  $M \xrightarrow{-}_{bxgc} N$  and  $M, N$  pure. We will do induction on the length of the  $\xrightarrow{-}_{bxgc}$ -reduction and prove

$$\begin{array}{ccccccc} M & \xrightarrow{-}_{bxgc} & M_1 & \xrightarrow{-}_{bxgc} & \cdots & \xrightarrow{-}_{bxgc} & M_{n-1} & \xrightarrow{-}_{bxgc} & N \\ \parallel & & \downarrow x & & & & \downarrow x & & \parallel \\ M & \xrightarrow{-}_b & \downarrow_x(M_1) & \xrightarrow{-}_b & \cdots & \xrightarrow{-}_b & \downarrow_x(M_{n-1}) & \xrightarrow{-}_b & N \end{array}$$

Each step in the top of each ‘cell’ is one of  $\xrightarrow{-}_{gc}$ ,  $\xrightarrow{-}_x$ , and  $\xrightarrow{-}_b$ , thus each ‘cell’ is an instance of either Proposition 3.1.13.A, confluence of  $\xrightarrow{-}_x$ , or Lemma 3.1.16.A.  $\square$

An easy consequence of this is the following.

**3.1.18 Corollary.**  $\xrightarrow{-}_{bxgc} \rightarrow CR$ .

*Proof.* By Proposition 2.1.17.A with  $\xrightarrow{-}_x$  as projection.  $\square$

**3.1.19 Discussion (locality).** We argued in the introduction that explicit substitution would establish a *complexity faithful* method of reducing  $\lambda$ -terms. This is manifested in the definition in the fact that *contraction creates a bounded fragment of the contractum  $\lambda x$ -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 5.

## 3.2 Preservation of Strong Normalisation

In this section we prove preservation of strong normalisation of untyped  $\lambda$ -terms (PSN) for  $\lambda xgc$ . We first recall Melliès's (1995) counterexample to PSN of  $\beta$ -reduction for  $\lambda\sigma$  of Abadi et al. (1991)<sup>1</sup> 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  $\beta$ -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  $\lambda xgc$ -reduction, and show that it is a confluent refinement of the  $\lambda$ -calculus, with  $\beta$ -PSN. Afterwards we show that the 'garbage collection overhead' does not provide for infinite reductions from which we conclude that unrestricted  $\lambda xgc$ -calculus has PSN. Our proof is direct. Furthermore, by reasoning more carefully and avoiding infinite reduction sequences, the proof can be made constructive.

**3.2.1 Remark (composition of substitution breaks PSN).** Why not consider a stronger calculus with some kind of reduction rule for composition of substitutions as  $\lambda\sigma$  has? It seems poor that for  $\lambda xgc$  the substitution lemma is only an extensional property of reduction whereas for  $\lambda\sigma$  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.  $\lambda\sigma$  essentially adds *parallel substitution* which in our notation could be written  $\langle \vec{x} := \vec{N} \rangle$  which intuitively means substitute  $N_1$  for  $x_1$ ,  $N_2$  for  $x_2$ ,  $\dots$ ,  $N_m$  for  $x_m$  simultaneously; the  $\lambda\sigma$  rules *Clos* and *Map* are equivalent to the parallel substitution rule

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

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

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

suffices for an exposition of Melliès's counterexample as shown in Figure 3.3 (the original term was the typable  $\lambda x.(\lambda y.I(Iy))(Ix)$ ).

<sup>1</sup>We present  $\lambda\sigma$  in Definition 3.3.21.

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

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

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

$$\begin{aligned} & (\lambda y. (\lambda y'. a) ((\lambda y. a) b)) ((\lambda y. a) b) \\ & \rightarrow a \langle y' := (\lambda y. a) b \rangle \langle y := (\lambda y. a) b \rangle \\ & \rightsquigarrow a \langle y' := ((\lambda y. a) b) \langle y := (\lambda y. a) b \rangle \rangle \\ & \rightarrow a \langle y' := (\lambda y. a \langle y := (\lambda y. a) b \rangle) (b \langle y := (\lambda y. a) b \rangle) \rangle \\ & \equiv a \langle y' := (\lambda y. a S_0) (b S_0) \rangle \\ & \rightarrow a \langle y' := a S_0 \langle y := b S_0 \rangle \rangle \\ & \equiv a \langle y' := a S_0 S_1 \rangle, \end{aligned}$$

$$\begin{aligned} a S_0 S_{m+1} & \equiv a \langle y := (\lambda y. a) b \rangle \langle y := b S_m \rangle \equiv a \langle y' := (\lambda y. a) b \rangle \langle y := b S_m \rangle \\ & \rightsquigarrow a \langle y' := ((\lambda y. a) b) \langle y := b S_m \rangle \rangle \\ & \rightarrow a \langle y' := (\lambda y. a \langle y := b S_m \rangle) (b \langle y := b S_m \rangle) \rangle \\ & \equiv a \langle y' := (\lambda y. a S_{m+1}) (b S_{m+1}) \rangle \\ & \rightarrow a \langle y' := a S_{m+1} \langle y := b S_{m+1} \rangle \rangle \\ & \equiv a \langle y := a S_{m+1} S_{m+2} \rangle, \end{aligned}$$

$$\begin{aligned} \text{and } a S_{m+1} S_{n+1} & \equiv a \langle y' := b S_m \rangle \langle y := b S_n \rangle \\ & \rightsquigarrow a \langle y' := b S_m \langle y := b S_n \rangle \rangle \equiv a \langle y' := b S_m S_{n+1} \rangle \end{aligned}$$

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

$$\begin{aligned} & (\lambda y. (\lambda y'. a) ((\lambda y. a) b)) ((\lambda y. a) b) \rightarrow \dots S_0 S_1 \dots \rightarrow \dots S_1 S_2 \dots \\ & \rightarrow \dots S_0 S_2 \dots \rightarrow \dots S_2 S_3 \dots \rightarrow \dots S_1 S_3 \dots \\ & \rightarrow \dots S_0 S_3 \dots \rightarrow \dots S_3 S_4 \dots \rightarrow \dots \\ & \vdots \end{aligned}$$

Figure 3.3: Simplified version of Mellès's counterexample to PSN.

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  $\lambda xgc$ , 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  $\beta$ -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  $\xrightarrow{gc}$ : it allows us to attack the question whether  $\xrightarrow{bxgc}$  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.

**3.2.2 Definition (garbage-free reduction).**  $\xrightarrow{bx\downarrow gc}$  is  $(\xrightarrow{x} \cdot \xrightarrow{gc}^*) \cup (\xrightarrow{b} \cdot \xrightarrow{gc}^*)$ , *i.e.*, the union of the composition of each of  $\xrightarrow{x}$  and  $\xrightarrow{b}$  with complete gc-reduction. We denote 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 3.1.15).

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  $\xrightarrow{gc}^*$ -reduction to gc-nf.

**3.2.3 Lemmas.**

A. For all  $\lambda x$ -terms  $M$ ,

$$\begin{array}{ccc} M & \xrightarrow{bxgc} & N \\ \downarrow gc & & \downarrow gc \\ \downarrow gc(M) & \xrightarrow{bx\downarrow gc} & \downarrow gc(N) \end{array} .$$

B. For garbage-free  $M$ ,

$$M \xrightarrow{\text{bxgc}} N \xrightarrow{\text{gc}} \Downarrow_{\text{gc}}(N)$$

$$\Downarrow_{\text{gc}}(M) \xrightarrow{\text{bx}\downarrow\text{gc}} \Downarrow_{\text{gc}}(N)$$

*Proof.* A by easy induction on the structure of  $M$ . B then follows by

$$M \xrightarrow{\text{bxgc}} M_1 \xrightarrow{\text{gc}} M_2 \xrightarrow{\text{bxgc}} M_3 \xrightarrow{\text{gc}} M_4 \xrightarrow{\text{bxgc}} \cdots \xrightarrow{\text{gc}} N$$

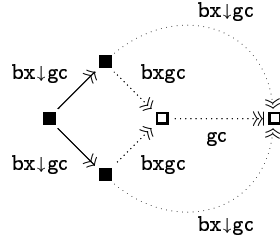
$$\Downarrow_{\text{gc}}(M_1) \equiv \Downarrow_{\text{gc}}(M_2) \xrightarrow{\text{bx}\downarrow\text{gc}} \Downarrow_{\text{gc}}(M_3) \equiv \Downarrow_{\text{gc}}(M_4) \xrightarrow{\text{bx}\downarrow\text{gc}} \cdots \equiv \Downarrow_{\text{gc}}(N)$$

(\*)

using A several times. □

### 3.2.4 Theorem. $\xrightarrow{\text{bx}\downarrow\text{gc}} \text{CR}$ .

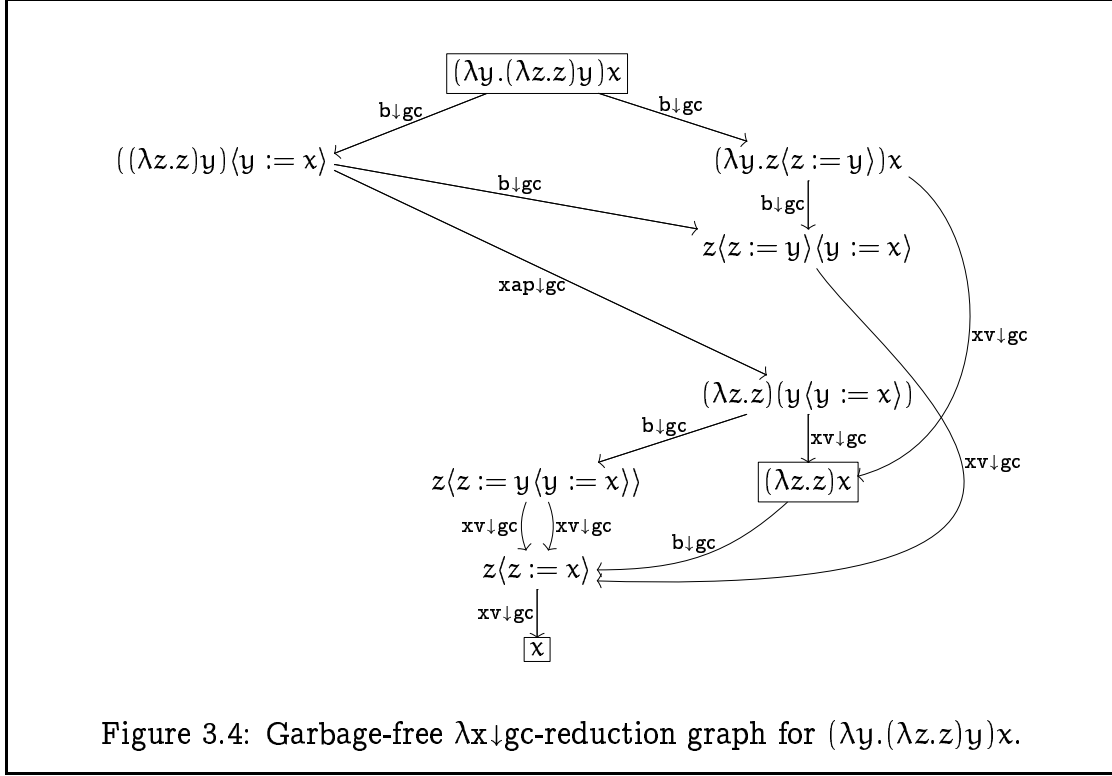
*Proof.* We show  $\xrightarrow{\text{bx}\downarrow\text{gc}} \diamond$ : it follows from  $\xrightarrow{\text{bx}\downarrow\text{gc}} \subseteq \xrightarrow{\text{bxgc}} \gg$ , the confluence of  $\xrightarrow{\text{bxgc}}$ , and (two applications of) Lemma 3.2.3.B by



□

**3.2.5 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  $v\langle w := x \rangle \langle y := z \rangle \xrightarrow{\text{gc}} v\langle w := x \rangle \xrightarrow{\text{gc}} v$ . 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 3.1.8, it has the garbage-free reduction graph depicted in Figure 3.4 (compare to Figure 3.2). Note that the only place in this graph where a single garbage-free step consists of more than one raw step is where  $((\lambda z. z)y)\langle y := x \rangle \xrightarrow{\text{xap}} ((\lambda z. z)\langle y := x \rangle)(y\langle y := x \rangle) \xrightarrow{\text{gc}} (\lambda z. z)(y\langle y := x \rangle)$ .

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



*Proof.* We will show that each  $\xrightarrow{\text{bx} \downarrow_{gc}}$ -reduction corresponds to a  $\xrightarrow{\beta}$ -reduction of comparable length, hence there are no infinite  $\xrightarrow{\text{bx} \downarrow_{gc}}$ -reductions for terms that are strongly normalising for  $\xrightarrow{\beta}$ ; the proof is similar to that of Theorem 3.1.17. Assume  $M$  is pure and strongly normalising for  $\xrightarrow{\beta}$ . Since  $M$  is pure it has no  $\xrightarrow{xgc}$ -redexes and every  $\xrightarrow{\text{bx} \downarrow_{gc}}$ -reduction starting with  $M$ , whether finite or not, is of the form  $M \equiv M_0 \xrightarrow{b} M_1 \xrightarrow{xgc} M_2 \xrightarrow{b} M_3 \xrightarrow{xgc} \dots$  where the “ $\xrightarrow{xgc}$ ” segments are really  $\xrightarrow{gc} \cdot (\xrightarrow{x} \cdot \xrightarrow{gc}) \cdot \dots \cdot (\xrightarrow{x} \cdot \xrightarrow{gc})$  sequences. From such a sequence we can construct a  $\xrightarrow{\beta}$ -sequence as follows, from left to right:

$$\begin{array}{ccccccc}
 M \equiv M_0 & \xrightarrow{b} & M_1 & \xrightarrow{xgc} & M_2 & \xrightarrow{b} & M_3 \xrightarrow{xgc} \dots \\
 \parallel & & \begin{array}{c} (*) \\ \vdots \\ x \end{array} & & \begin{array}{c} (**) \\ \vdots \\ x \end{array} & & \begin{array}{c} (*) \\ \vdots \\ x \end{array} & & \begin{array}{c} (**) \\ \vdots \\ x \end{array} \\
 \downarrow_x(M_0) & \xrightarrow{\beta} & \downarrow_x(M_1) & \dots & \downarrow_x(M_2) & \xrightarrow{\beta} & \downarrow_x(M_3) & \dots
 \end{array}$$

where  $(*)$  is Lemma 3.1.16.B and  $(**)$  is Proposition 3.1.13.A. Since the lower reduction is finite and  $\xrightarrow{xgc} \text{SN}$ , 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.

**3.2.7 Definitions.** We subdivide the reduction relation  $\xrightarrow{\text{bxgc}}$  into two mutually disjoint parts.

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

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

Note the use of  $\text{fv}(\downarrow_{\text{gc}}(\cdot))$  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.

B. 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  $\xrightarrow{\text{gc}}$ -normalform.

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

### 3.2.8 Propositions.

- A. If  $M \xrightarrow{\text{bxgc}} N$  is garbage-reduction then  $\downarrow_{\text{gc}}(M) \equiv \downarrow_{\text{gc}}(N)$ .
- B. If  $M \xrightarrow{\text{bxgc}} N$  is outside garbage then  $\downarrow_{\text{gc}}(M) \xrightarrow{\text{bx}\downarrow_{\text{gc}}} \downarrow_{\text{gc}}(N)$ .

**3.2.9 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  $\text{subSN}(M)$  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.



### 3.2.10 Lemmas.

- A. If  $\text{subSN}(M)$  and  $M \xrightarrow{\text{bxgc}} N$  is garbage-reduction, then  $\text{subSN}(N)$ .
- B. If  $\text{subSN}(M)$  then  $M$  is strongly normalising for garbage-reduction.

*Proof.* A Induction on the structure of  $M$ . We treat the case  $M\langle x := N \rangle \xrightarrow{\text{bxgc}} M\langle x := N' \rangle$  where  $x \notin \text{fv}(\downarrow_{\text{gc}}(M))$ . 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  $\text{subSN}(M\langle x := N \rangle)$ .

B Define two interpretations for subSN-terms  $M$ . For terms  $M$  such that  $\text{subSN}(M)$ , 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  $\text{subSN}(M)$ ,  $h_1(M)$  exists. Define  $h_2$  by the following:

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

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.  $\square$

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

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

*Proof.* We use induction on  $\#gf(M)$ .

**Base case**  $\#gf(M) = 0$ . Then by Proposition 3.2.8.B reduction paths of  $M$  cannot contain reductions outside garbage and hence by Lemma 3.2.10.B they are finite.

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

$$M \equiv M_0 \xrightarrow{\text{bxgc}} M_1 \xrightarrow{\text{bxgc}} M_2 \xrightarrow{\text{bxgc}} M_3 \xrightarrow{\text{bxgc}} \dots$$

By Lemma 3.2.10.B there is  $m$  such that  $M \xrightarrow{\text{bxgc}} M_m$  is garbage-reduction and  $M_m \xrightarrow{\text{bxgc}} M_{m+1}$  is reduction outside garbage. Then by Proposition 3.2.8,  $\#gf(M_{m+1}) < \#gf(M_m)$  and by Lemma 3.2.10.A,  $\text{subSN}(M_m)$ . Now we prove by induction on the structure of  $M_m$  that also  $\text{subSN}(M_{m+1})$ ; then by IH1 we are done. Call the new induction hypothesis IH2. We treat some typical cases:

**Case**  $M_m \equiv (N_1 N_2) \langle x := P \rangle \xrightarrow{\text{bxgc}} (N_1 \langle x := P \rangle) (N_2 \langle x := P \rangle) \equiv M_{m+1}$ , where  $x \in \text{fv}(\downarrow_{\text{gc}}(N_1 N_2))$ . 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  $(N_1 N_2) \langle x := P \rangle$  and  $\text{subSN}((N_1 N_2) \langle x := P \rangle)$ .

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

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

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

**3.2.13 Corollary (PSN for  $\lambda xgc$ ).**  $\xrightarrow{\text{bxgc}}$  PSN of  $\xrightarrow{\beta}$ .

*Proof.* **Case  $\Rightarrow$ .** If  $M$  is pure then  $\text{subSN}(M)$  and if  $M$  is strongly normalising for  $\xrightarrow{\beta}$ -reduction then by Theorem 3.2.6,  $\#gf(M) < \infty$ . Now use Theorem 3.2.12.

**Case  $\Leftarrow$ .** By Proposition 3.1.15.B, infinite  $\xrightarrow{\beta}$ -reductions induce infinite  $\xrightarrow{\text{bxgc}}$ -reductions.  $\square$

**3.2.14 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 Theorem 3.3.9.

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

**3.2.15 Corollary.** *A  $\lambda xgc$ -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 3.2.13.

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,  $\text{subSN}(M)$ . After one garbage-free reduction step  $M \xrightarrow{\text{bx}\downarrow_{gc}} M'$ , 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 3.2.6. Therefore,  $\#gf(M) < \infty$ . Now by Theorem 3.2.12,  $M$  is SN for  $\xrightarrow{\text{bxgc}}$ -reduction.  $\square$

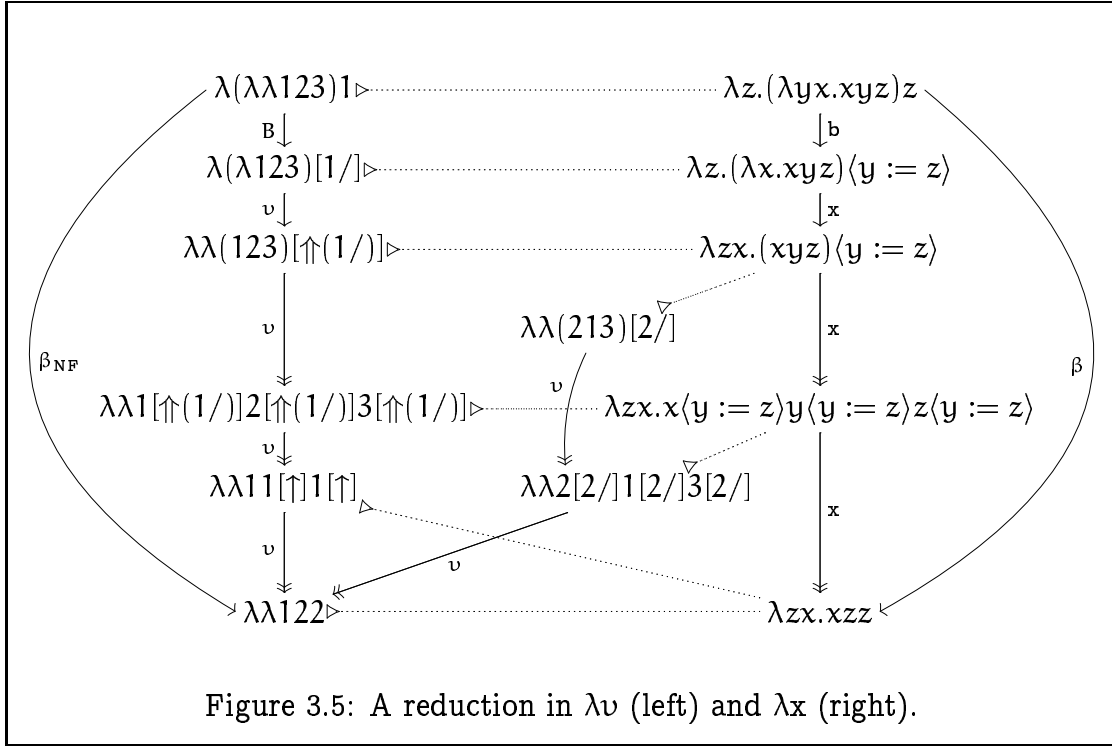
**3.2.16 Remark.** Finally we mention a positive result: adding the rewrite rule

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

does not break PSN; see Bloo and Geuvers (1996) for details. Intuitively the reason for this is that if  $x \in \text{fv}(\downarrow_{gc}(M))$  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 3.2.1 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.

### 3.3 Explicit Substitution & Naming

In this section we combine the notion of explicit naming as presented in section 2.5 with explicit substitution and show how this assigns to traditional calculi



of explicit substitution a set of ‘coordinates’ in the explicit naming and explicit substitution dimensions.

We commence with a preliminary study of the naming dimension by investigating in detail the relation between  $\lambda x$  and the simplest de Bruijn-index based calculus of explicit substitution that we know, namely  $\lambda\nu$  of Lescanne (1994a). We will give a translation from  $\lambda x$  to  $\lambda\nu$  which is sufficiently strong that we can provide an independent and direct proof of PSN for  $\lambda\nu$ . We then abstract a sufficient condition on such translations, called *strict explicit naming* which guarantees PSN, and we finally use this to give new and direct proofs of several known PSN results: for  $\lambda s$  of Kamareddine and Ríos (1995) and  $\lambda\chi$  of Lescanne and Rouyer-Degli (1995).

**3.3.1 Discussion (comparison between  $\lambda\nu$  and  $\lambda x$ ).** The  $\lambda$ -term  $\lambda z.(\lambda y x. x y z) z$  corresponds to the de Bruijn term  $\lambda(\lambda\lambda 123)1$ . This term reduces with  $\lambda\nu$  and  $\lambda x$  as shown in Figure 3.5 ( $\xrightarrow{\beta_{NF}}$  is de Bruijn reduction of Definition 2.5.1). The used translation is suggested by  $\triangleright$ ; notice how some terms have more reduction possibilities in  $\lambda\nu$  than in  $\lambda x$  – in fact each  $\lambda x$ -term corresponds to a class of  $\lambda\nu$ -terms with a common  $v$ -reduct, as we shall see.

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

$$a ::= \underline{n} \mid \lambda a \mid ab \mid a[s] \quad (\text{terms})$$

$$s ::= a/ \mid \uparrow(s) \mid \uparrow \quad (\text{substitutions})$$

where the letters  $abc$  range over terms,  $st$  over substitutions, and  $nm$  over natural numbers.

**Substitution generation.**  $\xrightarrow{\mathbb{B}}$  is the contextual closure of

$$(\lambda a)b \rightarrow a[b/] \quad (\text{B})$$

**Substitution elimination.**  $\xrightarrow{\mathbb{V}}$  is the contextual closure of the union of

$$(ab)[s] \rightarrow a[s] b[s] \quad (\text{App})$$

$$(\lambda a)[s] \rightarrow \lambda(a[\uparrow(s)]) \quad (\text{Lambda})$$

$$\underline{1}[a/] \rightarrow a \quad (\text{FVar})$$

$$\underline{n+1}[a/] \rightarrow \underline{n} \quad (\text{RVar})$$

$$\underline{1}[\uparrow(s)] \rightarrow \underline{1} \quad (\text{FVarLift})$$

$$\underline{n+1}[\uparrow(s)] \rightarrow \underline{n}[s][\uparrow] \quad (\text{RVarLift})$$

$$\underline{n}[\uparrow] \rightarrow \underline{n+1} \quad (\text{VarShift})$$

**Reduction.**  $\lambda\nu$ -reduction,  $\xrightarrow{\mathbb{B}\mathbb{V}}$ , is the union of  $\xrightarrow{\mathbb{B}}$  and  $\xrightarrow{\mathbb{V}}$ .

Figure 3.6:  $\lambda\nu$ .

**3.3.2 Definition ( $\lambda\nu$ ).** See Figure 3.6.

Notice how this is just one way of making the substitution of Definition 2.5.1 explicit:  $\lambda\nu$ 's  $\_/\_$  corresponds to  $\_ \cdot \iota$ ,  $\uparrow(-)$  to  $1 \cdot (\iota/1 \circ \_)$ , and  $\uparrow$  to  $\iota/1$ .

**3.3.3 Proposition.**  $\xrightarrow{\nu} \text{SN}$ .

*Proof.* In Benaissa, Briaud, Lescanne and Rouyer-Degli (1995, Figure 3).  $\square$

**3.3.4 Remark.** We will not discuss garbage collection and thus only use the  $\lambda x$  subrelation of  $\lambda xgc$  as presented in section 3.1. Including garbage collection reductions would complicate things further in that it is not so easy to express the notion of 'garbage' when there are no implicit binding relation and hence no observable notion of free variables!

As the diagram illustrates, it is fairly straightforward to translate  $\lambda\nu$ -terms to  $\lambda x$ -terms by generalising the generic de Bruijn translation of Definition 2.5.4 to get rid of the  $\uparrow$  and  $\uparrow\uparrow$  constructions, essentially emulating their effect.

**3.3.5 Definition (translation,  $\lambda\nu/\lambda x$ ).** For an  $\lambda\nu$ -term  $a$  and a  $\lambda x$ -term  $M$  we write  $a \triangleright_{\nu} M$  iff  $\epsilon \vdash a \triangleright_{\nu} M$ , where

$$\begin{array}{c} \rho \vdash n \triangleright_{\nu} x \quad \rho(n) = x \\ \\ \frac{x \cdot \rho \vdash a \triangleright_{\nu} M}{\rho \vdash \lambda a \triangleright_{\nu} \lambda x.M} \quad x \notin \rho \\ \\ \frac{\rho \vdash a \triangleright_{\nu} M \quad \rho \vdash b \triangleright_{\nu} N}{\rho \vdash ab \triangleright_{\nu} MN} \\ \\ \frac{\rho \setminus n \cdot x \cdot \rho/n \vdash a \triangleright_{\nu} M \quad \rho/n \vdash b \triangleright_{\nu} N}{\rho \vdash a[\uparrow^n(b/)] \triangleright_{\nu} M\langle x := N \rangle} \quad x \notin \rho, n \geq 0 \\ \\ \frac{\rho \setminus n \cdot \rho/(n+1) \vdash a \triangleright_{\nu} M}{\rho \vdash a[\uparrow^n(\uparrow)] \triangleright_{\nu} M} \quad n \geq 0 \end{array}$$

( $\rho$  is a sequence of variables in the sense of Notation 2.2.1).

**3.3.6 Remark.** A different choice of the variable of each abstraction/substitution gives an  $\alpha$ -equivalent term as a result. Also note how  $\uparrow$  with a shift substitution removes a variable from the scope whereas similarly  $\uparrow$  with a real substitution inserts one.

This is a translation as one should expect; however, we can easily show the stronger result that the substitution introduction and elimination subreductions of  $\lambda\nu$  correspond closely to those of  $\lambda x$  – in fact, since our goal is to prove PSN we are particularly careful to get a strict correspondence between the substitution introduction relations  $\xrightarrow{\mathbb{B}}$  and  $\xrightarrow{\overline{\mathbb{B}}}$  in the sense that it is *not* only a property of the transitive closures:

- 3.3.7 Propositions.**
- A.  $\triangleright_{\nu}$  is a projection function of  $\xrightarrow{\mathbb{B}\nu}$  into  $\xrightarrow{\text{bxgc}}$ .
  - B.  $\triangleright_{\nu}$  is a projection function of  $\xrightarrow{\mathbb{B}}$  into  $\xrightarrow{\overline{\mathbb{B}}}$ .
  - C.  $\triangleright_{\nu}$  is a projection function of  $\xrightarrow{\nu}$  into  $\xrightarrow{\overline{\nu}}$ .

*Proof.* First of all, it is easy to see from the definition that  $\triangleright_{\nu}$  is a function in all three cases (*cf.* Remark 3.3.6). For this reason we will in this proof use the notation  $\overline{a\rho}$  for the unique  $\lambda x$ -term  $M$  satisfying  $\rho \vdash a \triangleright_{\nu} M$ .

A is clearly a direct consequence of B and C which we will prove by induction over the structure of  $\lambda\nu$ -terms, using the definition to investigate each  $\lambda\nu$ -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  $\rho$  that  $\overline{a\rho} \rightarrow \overline{b\rho}$ . B 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 C we include all the base cases as it is instructive to see what happens, in particular this is very similar to the reasoning in Benaissa, Briaud, Lescanne and Rouyer-Degli (1995, section 3):

Case (App) with real substitution.

$$\begin{aligned} \overline{(ab)[\uparrow^n(b/)] \rho} &\equiv \overline{(ab) ((\rho \setminus n) \cdot (x) \cdot (\rho/n))} \langle x := \overline{b(\rho/n)} \rangle \\ &\equiv \overline{(a((\rho \setminus n) \cdot (x) \cdot (\rho/n)) \overline{b((\rho \setminus n) \cdot (x) \cdot (\rho/n))})} \langle x := \overline{b(\rho/n)} \rangle \\ &\xrightarrow{\overline{x}} \overline{a((\rho \setminus n) \cdot (x) \cdot (\rho/n))} \langle x := \overline{b(\rho/n)} \rangle \\ &\quad \overline{b((\rho \setminus n) \cdot (x) \cdot (\rho/n))} \langle x := \overline{b(\rho/n)} \rangle \\ &\equiv \overline{a[\uparrow^n(b/)] \rho} \overline{b[\uparrow^n(b/)] \rho} \end{aligned}$$

Case (App) with shift-substitution.

$$\begin{aligned} \overline{(\mathbf{ab})[\uparrow^n]} \rho &\equiv \overline{(\mathbf{ab}) ((\rho \setminus \mathbf{n}) \cdot (\rho/\mathbf{n} + 1))} \\ &\equiv \overline{\mathbf{a} ((\rho \setminus \mathbf{n}) \cdot (\rho/\mathbf{n} + 1)) \mathbf{b} ((\rho \setminus \mathbf{n}) \cdot (\rho/\mathbf{n} + 1))} \\ &\equiv \overline{\mathbf{a}[\uparrow^n]} \rho \overline{\mathbf{a}[\uparrow^n]} \rho \end{aligned}$$

Case (Lambda) with real substitution.

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

Case (Lambda) with shift-substitution.

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

Case (FVar).  $\overline{\mathbf{1}[\mathbf{a}/]} \rho \equiv \overline{\mathbf{1}((\mathbf{x}) \cdot \rho)} \langle \mathbf{x} := \overline{\mathbf{a}\rho} \rangle \equiv \mathbf{x} \langle \mathbf{x} := \overline{\mathbf{a}\rho} \rangle \xrightarrow{\mathbf{x}} \overline{\mathbf{a}\rho}$ .

Case (RVar).  $\overline{\mathbf{n} + \mathbf{1}[\mathbf{a}/]} \rho = \overline{\mathbf{n} + \mathbf{1}((\mathbf{x}) \cdot \rho)} \langle \mathbf{x} := \overline{\mathbf{a}\rho} \rangle \equiv \rho(\mathbf{n}) \langle \mathbf{x} := \overline{\mathbf{a}\rho} \rangle \xrightarrow{\mathbf{x}} \rho(\mathbf{n}) \equiv \overline{\mathbf{n}\rho}$ .

Case (FVarLift) with real substitution.

$$\begin{aligned} \overline{\mathbf{1}[\uparrow^{n+1}(\mathbf{a}/)]} \rho &\equiv \overline{\mathbf{1}((\rho \setminus \mathbf{n} + 1) \cdot (\mathbf{x}) \cdot (\rho/\mathbf{n} + 1)) \langle \mathbf{x} := \overline{\mathbf{a}\rho} \rangle} \\ &\equiv \rho(1) \langle \mathbf{x} := \overline{\mathbf{a}\rho} \rangle \xrightarrow{\mathbf{x}} \rho(1) \equiv \overline{\mathbf{1}\rho} \end{aligned}$$

Case (FVarLift) with shift-substitution.

$$\overline{\mathbf{1}[\uparrow^{n+1}(\uparrow)]} \rho \equiv \overline{\mathbf{1}((\rho \setminus \mathbf{n} + 1) \cdot (\rho/\mathbf{n} + 2))} \equiv \rho(1) \equiv \overline{\mathbf{1}\rho}$$



Case (RVarLift) with real substitution.

$$\begin{aligned}
\overline{\mathfrak{m} + 1[\uparrow^{n+1}(\mathfrak{a}/)]} \rho &\equiv \overline{\mathfrak{m} + 1((\rho \setminus \mathfrak{n} + 1) \cdot (\mathfrak{x}) \cdot (\rho/\mathfrak{n} + 1))\langle \mathfrak{x} := \overline{\mathfrak{a}\rho/\mathfrak{n} + 1} \rangle} \\
&\equiv \overline{\mathfrak{m}((\rho/1 \setminus \mathfrak{n}) \cdot (\mathfrak{x}) \cdot (\rho/\mathfrak{n} + 1))\langle \mathfrak{x} := \overline{\mathfrak{a}(\rho/\mathfrak{n} + 1)} \rangle)} \\
&\equiv \overline{\mathfrak{m}[\uparrow^n(\mathfrak{a}/)]} (\rho/1) \\
&\equiv \overline{\mathfrak{m}[\uparrow^n(\mathfrak{a}/)]}[\uparrow] \rho
\end{aligned}$$

Case (RVarLift) with shift-substitution.

$$\begin{aligned}
\overline{\mathfrak{m} + 1[\uparrow^{n+1}(\uparrow)]} \rho &\equiv \overline{\mathfrak{m} + 1((\rho \setminus \mathfrak{n} + 1) \cdot (\rho/\mathfrak{n} + 2))} \\
&\equiv \begin{cases} \rho(\mathfrak{m} + 1) & \text{if } \mathfrak{m} \leq \mathfrak{n} \\ \rho(\mathfrak{m} + 2) & \text{if } \mathfrak{m} > \mathfrak{n} \end{cases} \\
&\equiv \overline{\mathfrak{m}((\rho/1 \setminus \mathfrak{n}) \cdot (\rho/\mathfrak{n} + 2))} \\
&\equiv \overline{\mathfrak{m}[\uparrow^n(\uparrow)]} (\rho/1) \\
&\equiv \overline{\mathfrak{m}[\uparrow^n(\uparrow)]}[\uparrow] \rho
\end{aligned}$$

Case (VarShift).  $\overline{\mathfrak{n}[\uparrow]} \rho \equiv \overline{\mathfrak{n}} \rho/1 \equiv \rho(\mathfrak{n} + 1) \equiv \overline{\mathfrak{n} + 1} \rho$ .

Note that the  $\lambda\nu$ -reductions that correspond to nothing on  $\lambda\mathfrak{x}$ -terms after translations are: (Lambda) and (App) with  $s = \uparrow$ , (FVarLift) and (RVarLift), and (VarShift).  $\square$

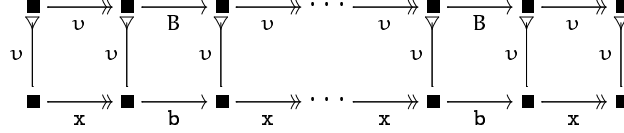
The above suffices to show the desired properties of  $\lambda\nu$ , namely that it is a conservative extension of the  $\lambda\beta$ -calculus that preserves strong normalisation:

**3.3.8 Theorem (conservative extension,  $\lambda\nu$  of  $\lambda\beta$ ).** *Given  $\lambda\nu$ -terms  $\mathfrak{a}, \mathfrak{b}$  and pure  $\lambda$ -terms  $\mathfrak{M}, \mathfrak{N}$  such that  $\mathfrak{a} \triangleright_{\nu} \cdot \xrightarrow{\mathfrak{x}} \mathfrak{M}$  and  $\mathfrak{b} \triangleright_{\nu} \cdot \xrightarrow{\mathfrak{x}} \mathfrak{N}$ . Then  $\mathfrak{a} \xrightarrow{\text{Bv}} \mathfrak{b}$  if and only if  $\mathfrak{M} \xrightarrow{\beta} \mathfrak{N}$ .*

*Proof.* Follows directly from Proposition 3.3.7.A (original proof by Benaissa et al. 1995, sections 3 and 4).  $\square$

**3.3.9 Theorem (PSN  $\lambda\nu$ ).**  $\lambda\nu$  PSN of  $\lambda\beta$ .

*Proof.* We give a direct proof. Consider any  $\lambda\nu$ -reduction sequence. It must correspond to a  $\lambda x$ -sequence as follows by Proposition 3.3.7.B and Proposition 3.3.7.C:



This is thus finite exactly when the original sequence was finite because  $\xrightarrow{\nu}$ SN. Since  $\lambda x$  has PSN by Corollary 3.2.13 we are done. (Original indirect proof by Benaissa et al. 1995, section 5).  $\square$

**3.3.10 Comparison (on minimal derivations).** Comparing this proof with the original proof of Benaissa et al. is interesting. The translation to  $\lambda x$  has eliminated the need for considering ‘minimal derivations’ and positions that are ‘internal/external’ to substitutions. In the PSN proof for  $\lambda x$  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  $\lambda\nu$  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 naming’. 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.

**3.3.11 Definition (explicit naming).** An *explicit naming principle* is a projection from a TRS into a reduction on a  $\lambda$ -calculus with names.

So Definition 2.5.4 and Definition 3.3.5 are naming principles. The crucial property for PSN is that there is a distinguished ‘ $\beta$ -like’ subreduction that is repeated as frequently as  $\beta$ -reduction. This we use to define a ‘strict explicit naming extension of  $\lambda x$ ’ which is what we need to obtain the desired PSN results.

**3.3.12 Definition (strict explicit naming).** The TRS  $\rightsquigarrow$  is a *strict explicit naming extension of  $\lambda x$*  if  $\rightsquigarrow = \rightsquigarrow_B \cup \rightsquigarrow$  such that the following conditions are satisfied:

- A.  $\triangleright$  is a projection from  $\rightsquigarrow$  into  $\xrightarrow{\text{bxgc}}$ .

B.  $\rightsquigarrow$ SN and  $\triangleright$  is a projection from  $\rightsquigarrow_x$  into  $\xrightarrow{x}$ .

C.  $\triangleright$  is a projection from  $\rightsquigarrow$  into  $\xrightarrow{+}{B}$ .

**3.3.13 Theorem (PSN for strict explicit naming).** *If  $\rightsquigarrow$  is a strict explicit naming extension of  $\lambda x$ , then  $\rightsquigarrow$ PSN.*

First, it is clear that  $\lambda v$  is a strict explicit naming extension of  $\lambda x$ . Now all we have to do is apply this theorem to some other calculi. We proceed by giving translations for some calculi. First  $\lambda s$  of Kamareddine and Ríos (1995) where it is proven PSN by translation into  $\lambda v$  and with a minimal derivation proof mimicking the  $\lambda v$  one. Our translation below gives an independent, direct proof.

**3.3.14 Definition ( $\lambda s$ ).** The  $\lambda s$ -terms and  $\lambda s$ -reduction is defined in Figure 3.7.

This is another way of making the substitution notion of Definition 2.5.1 explicit: except for the ‘item’ infix style of explicit substitution (Kamareddine and Nederpelt 1993) it is very close to the notation of  $\lambda v$ ; this immediately suggests the projection to use: it is clearly the same for the pure  $\lambda$ NF-terms and slightly modified for the remaining.

**3.3.15 Definition (translation,  $\lambda s/\lambda x$ ).** For an  $\lambda s$ -term  $a$  and a  $\lambda x$ -term  $M$  we write  $a \triangleright_{\overline{s}} M$  iff  $\epsilon \vdash a \triangleright_{\overline{s}} M$ , where

$$\begin{array}{c} \rho \vdash n \triangleright_{\overline{s}} x \quad \rho(n) = x \\ \\ \frac{x \cdot \rho \vdash a \triangleright_{\overline{s}} M}{\rho \vdash \lambda a \triangleright_{\overline{s}} \lambda x.M} \quad x \notin \rho \\ \\ \frac{\rho \vdash a \triangleright_{\overline{s}} M \quad \rho \vdash b \triangleright_{\overline{s}} N}{\rho \vdash ab \triangleright_{\overline{s}} MN} \\ \\ \frac{\rho \setminus (i-1) \cdot x \cdot \rho / (i-1) \vdash a \triangleright_{\overline{s}} M \quad \rho / i \vdash b \triangleright_{\overline{s}} N}{\rho \vdash a \sigma^i b \triangleright_{\overline{s}} M \langle x := N \rangle} \quad x \notin \rho \\ \\ \frac{\rho \setminus k \cdot \rho / (k+i-1) \vdash a \triangleright_{\overline{s}} M}{\rho \vdash \phi_k^i a \triangleright_{\overline{s}} M} \end{array}$$

( $\rho$  is a sequence of variables).

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

$$a ::= n \mid ab \mid \lambda a \mid a \sigma^i b \mid \varphi_k^i a$$

where the letters  $abc$  range over terms, and  $ijk$  over natural numbers.

**Substitution generation.**  $\xrightarrow{\lambda s \sigma}$  is the contextual closure of

$$(\lambda a)b \rightarrow a \sigma^1 b \quad (\sigma\text{-gen})$$

**Substitution elimination.**  $\xrightarrow{s}$  is the contextual closure of the union of

$$(\lambda a)\sigma^i b \rightarrow \lambda(a \sigma^{i+1} b) \quad (\sigma\text{-}\lambda\text{-trans})$$

$$(a_1 a_2)\sigma^i b \rightarrow (a_1 \sigma^i b)(a_2 \sigma^i b) \quad (\sigma\text{-app-trans})$$

$$n \sigma^i b \rightarrow \begin{cases} n-1 & \text{if } n > i \\ \varphi_0^i b & \text{if } n = i \\ n & \text{if } n < i \end{cases} \quad (\sigma\text{-destr})$$

$$\varphi_k^i(\lambda a) \rightarrow \lambda(\varphi_{k+1}^i a) \quad (\varphi\text{-}\lambda\text{-trans})$$

$$\varphi_k^i(a_1 a_2) \rightarrow (\varphi_k^i a_1)(\varphi_k^i a_2) \quad (\varphi\text{-app-trans})$$

$$\varphi_k^i n \rightarrow \begin{cases} n+i-1 & \text{if } n > k \\ n & \text{if } n \leq k \end{cases} \quad (\varphi\text{-destr})$$

**Reduction.**  $\lambda s$ -reduction,  $\xrightarrow{\lambda s}$ , is the union of  $\xrightarrow{\lambda s \sigma}$  and  $\xrightarrow{s}$

Figure 3.7:  $\lambda s$ .

**3.3.16 Theorem.**  $\lambda_s$  is a strict explicit naming extension of  $\lambda_x$ .

*Proof.* Similar to the proof for  $\lambda_\nu$ . □

**3.3.17 Corollary (PSN  $\lambda_s$ ).**  $\lambda_s$  PSN of  $\lambda_\beta$ .

*Proof.* Immediate from being a strict explicit naming extension (original indirect proof by Kamareddine and Ríos 1995). □

Next  $\lambda_\chi$  of Lescanne and Rouyer-Degli (1995) which uses *de Bruijn-levels*.

**3.3.18 Definition ( $\lambda_\chi$ ).** The  $\lambda_\chi$ -calculus and  $\lambda_\chi$ -reduction is defined in Figure 3.8.

This is not directly related to de Bruijn's calculus, however, it *is* very close to  $\lambda_x$ , so the theorem is actually simpler to prove.

**3.3.19 Theorem.**  $\lambda_\chi$  is a strict explicit naming extension of  $\lambda_x$ .

*Proof.* Easy: the projection is just the rename-normal form since we can just reuse the variable names as  $\lambda_x$ -variables! □

**3.3.20 Corollary (PSN  $\lambda_\chi$ ).**  $\lambda_\chi$  PSN of  $\lambda_\beta$ .

(First, indirect, proof by Lescanne and Rouyer-Degli 1995)

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

**3.3.21 Definition ( $\lambda_\sigma$ ).** The  $\lambda_\sigma$ -calculus is defined in Figure 3.9.

Comparing this to de Bruijn's Definition 2.5.1, it is easy to see it simply makes *everything* explicit: the syntactic  $\_ \cdot \_$  and  $\_ \circ \_$  realise the similarly named sequence operators,  $\text{id}$  is our  $\iota$ , and  $\uparrow$  is  $\iota/1$ . The reason  $\lambda_\sigma$  does not preserve PSN, observed by Melliès (1995), is simple *interference* between the two subrelations  $\xrightarrow{\sigma}$  and  $\xrightarrow{\text{Beta}}$ : the substitution subrelation can accidentally create a Beta-redex (*cf.* Figure 3.3).

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

$$a ::= x_i \mid \lambda x_i \cdot a \mid ab \mid a[b/x_i]_j$$

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

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

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

**Substitution generation.**  $\xrightarrow{\mathbb{B}}$  is the contextual closure of

$$(\lambda x_i \cdot a)b \rightarrow a[b/x_i]_0 \quad (\mathbb{B})$$

**Substitution elimination.**  $\xrightarrow{\mathbb{X}}$  is the contextual closure of the union of

$$\begin{aligned} (a \ b)[c/x_i]_j &\rightarrow a[c/x_i]_j \ b[c/x_i]_j && (\text{App}) \\ (\lambda x_{i+j+1} \cdot a)[b/x_i]_j &\rightarrow \lambda x_{i+j} \cdot (a[b/x_i]_{j+1}) && (\text{Lambda}) \\ x_{i+k+1}[a/x_i]_j &\rightarrow x_{i+k} && (\text{Var}_{>}) \\ x_i[a/x_{i+k+1}]_j &\rightarrow x_i && (\text{Var}_{<}) \\ x_i[a/x_i]_j &\rightarrow \text{rename}(a, i, j) && (\text{Var}_{=}) \\ \text{rename}(ab, i, j) &\rightarrow \text{rename}(a, i, j) \ \text{rename}(b, i, j) && (\text{RenApp}) \\ \text{rename}(\lambda x_{i+k} \cdot a, i, j) &\rightarrow \lambda x_{i+k+j} \cdot \text{rename}(a, i, j) && (\text{RenLambda}) \\ \text{rename}(x_{i+k}, i, j) &\rightarrow x_{i+j+k} && (\text{RenVar}_{\geq}) \\ \text{rename}(x_i, i+k+1, j) &\rightarrow x_i && (\text{RenVar}_{<}) \end{aligned}$$

**Reduction.**  $\xrightarrow{\lambda\chi}$ , is the union of  $\xrightarrow{\mathbb{B}}$  and  $\xrightarrow{\mathbb{X}}$ .

Figure 3.8:  $\lambda\chi$

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

$$a ::= 1 \mid \lambda a \mid ab \mid a[s] \quad (\text{terms})$$

$$s ::= \text{id} \mid \uparrow \mid a \cdot s \mid s \circ t \quad (\text{substitutions})$$

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

**Substitution generation.**  $\xrightarrow{\text{Beta}}$  is the contextual closure of

$$(\lambda a)b \rightarrow a[b \cdot \text{id}] \quad (\text{Beta})$$

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

$$1[\text{id}] \rightarrow 1 \quad (\text{VarId})$$

$$1[a \cdot s] \rightarrow a \quad (\text{VarCons})$$

$$(ab)[s] \rightarrow a[s] b[s] \quad (\text{App})$$

$$(\lambda a)[s] \rightarrow \lambda(a[1 \cdot (s \circ \uparrow)]) \quad (\text{Abs})$$

$$(a[s])[t] \rightarrow a[s \circ t] \quad (\text{Clos})$$

$$\text{id} \cdot s \rightarrow s \quad (\text{IdL})$$

$$\uparrow \circ \text{id} \rightarrow \uparrow \quad (\text{ShiftId})$$

$$\uparrow \circ (a \cdot s) \rightarrow s \quad (\text{ShiftCons})$$

$$(a \cdot s) \circ t \rightarrow a[t] \cdot (s \circ t) \quad (\text{Map})$$

$$(s_1 \circ s_2) \circ s_3 \rightarrow s_1 \circ (s_2 \circ s_3) \quad (\text{Assoc})$$

Figure 3.9:  $\lambda\sigma$ .

**3.3.22 Discussion.** This section was mostly about how one can express existing de Bruijn calculi with explicit substitution by two components:  $\lambda x$  and a renaming principle. However, why not start with de Bruijn’s calculus as it is and make the substitution notion of Definition 2.5.1 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 2.5.1 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, understanding the naming dimension sufficiently to parametrise different namings more clearly – possibly the work of Gordon (1993) is appropriate to this end.

## 3.4 $\lambda$ -graphs and Explicit Sharing

A central theme to this dissertation is the notion of *sharing*. We present  $\lambda x_a$ , a  $\lambda$ -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*<sup>2</sup> and that of *term graph rewriting* (TGR) of Barendregt, van Eekelen, Glauert, Kennaway, Plasmeijer and Sleep (1987). TGR was based on the  $\lambda$ -graph reduction of Wadsworth (1971), and indeed the  $\lambda$ -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.

---

<sup>2</sup>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).



**3.4.1 Definitions (addressed terms).** Assume a set of terms  $T$  inductively defined by a single production  $t ::= \dots$ .

- A. The set of *addressed T-terms* is defined inductively by

$$t ::= \dots \mid t^a$$

where  $abc$  range over an infinite set of ‘addresses’.

- B. The *addresses* occurring in an addressed term is written  $\text{addr}(t)$ .
- C. *Unraveling* is written  $\Delta(t)$  and denotes the  $T$ -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.

**3.4.2 Definitions (sharing terms).** Assume a set of addressed  $T$ -terms as above.

- A. We write ‘ $t$  wfa’ if  $t$  is an addressed term where all subterms with the same addresses are identical, *i.e.*,

$$\forall t_1^{a_1}, t_2^{a_2} \triangleright t : a_1 = a_2 \implies t_1 = t_2$$

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

- B. The *updating* of  $t$  wfa with  $s$  wfa at address  $a \notin \text{addr}(s)$  is written  $t[s^a]$  and is obtained by replacing all subterms  $u^a \triangleright t$  (if any) by  $s^a$ . By convention  $t[\epsilon] \equiv t$  (to ensure  $t[t@_a] = t$  for all  $a$ ).
- C. The set of *sharing T-terms*, notation  $T$  wfa, is the subset of the addressed  $T$ -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  $a$ .

**3.4.3 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  $b$ s *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.

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

*Proof.* If  $c \notin \text{addr}(s)$  then  $s[t^c] = s$  wfa by definition. If  $c \in \text{addr}(s) \setminus \text{addr}(t)$  then each subterm  $u^c \triangleright s$  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.  $\square$

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.

**3.4.5 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

$\dashv\vdash$ , is the reduction on T wfa defined as follows: Given  $t \in \text{T wfa}$  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  $C[r] \dashv\vdash C[p]$  .

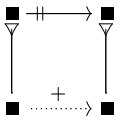
**Case**  $r$  is shared in  $C[r]$ : then it is possible to express  $C[r]$  on the form  $C_1[(C_2[r])^a]$  for some address  $a$ , where  $r$  is unshared in  $C_2[r]$  (*i.e.*,  $C_2[]$  contains no addresses) . Then

$$t \dashv\vdash (C_1[(C_2[p])^a]) [(C_2[p])^a]$$

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  $\dashv\vdash$  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:

**3.4.6 Proposition.** For all reductions  $\rightarrow$ , , where  $\triangleright$  is the relation defined by  $t \triangleright \Delta(t)$ .

*Proof.* Assume  $t \dashv\vdash s$ . Then from Definition 3.4.5 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  $a$ ,  $\Delta(t) \dashv\vdash \Delta(s)$ .  $\square$

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  $\lambda$ -graphs below.

**3.4.7 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

$$A(1,2) \rightarrow 0 \quad 1 \rightarrow 2 \quad 2 \rightarrow 1 \quad D(x) \rightarrow A(x,x)$$

includes the reduction

$$D(1) \rightarrow A(1,1) \rightarrow A(1,2) \rightarrow 0$$

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

$$\begin{array}{ccccccc} D & & A & & A & & A \\ \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow & \rightarrow & \downarrow \\ 1 & & 1 & & 2 & & 1 \end{array} \rightarrow \dots$$

This exists equivalently for our sharing extension  $\rightsquigarrow$  of  $\rightarrow$  where it is expressed as

$$D(1) \rightsquigarrow A(1^a, 1^a) \rightsquigarrow A(2^a, 2^a) \rightsquigarrow A(1^a, 1^a) \rightsquigarrow \dots$$

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

$$\begin{array}{c} A \\ \downarrow \\ x \end{array} \rightarrow \begin{array}{c} A \\ \swarrow \downarrow \\ x \quad x \end{array}$$

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  $\lambda$ -graph rewriting.

### 3.4.8 Definitions ( $\lambda a$ -terms).

- A. The set of  $\lambda a$ -preterms is the set of sharing  $\lambda$ -preterms.
- B. The notion of *free variables* for  $\lambda a$ -preterms is the one for  $\lambda$ -preterms (Definition 2.3.2.A) generalised to any  $\lambda a$ -preterm  $M$  by the equation

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

- C. *Renaming* for  $\lambda a$ -preterms is as for  $\lambda$ -preterms (Definition 2.3.2.B) generalised to any  $\lambda a$ -preterm  $M$  by

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

- D.  $\alpha$ -*equivalence* for  $\lambda a$ -preterms is as for  $\lambda$ -preterms (Definition 2.3.2.C) generalised to any  $\lambda a$ -preterm  $M$  by the equation

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

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

- E. The set of  $\lambda a$ -terms, is  $\Lambda a = \Lambda \text{ wfa}$ , is the  $\lambda a$ -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  $\beta$ -reduction naïvely to  $\lambda$ -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  $\lambda a$ -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  $\beta$ -reduction (the redex is underlined):

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

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,

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

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 corresponds to copying the subterm at  $a$  into as many copies as there are instances):

$$\begin{array}{c} \frac{(\lambda x. wx(\lambda z. zv)^b)^a}{\text{cp}} \quad ((\lambda x. wx(\lambda z. zv)^b)^a (\lambda z. zv)^b) \\ \downarrow \\ (\lambda x. wx(\lambda z. zv)^b) \quad ((\lambda x. wx(\lambda z. zv)^b) (\lambda z. zv)^b) \\ \downarrow \\ w((\lambda x. wx(\lambda z. zv)^b) (\lambda z. zv)^b) (\lambda z. zv)^b \end{array}$$

where we have used  $\xrightarrow{\text{cp}}$  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.

**3.4.9 Definitions (admissibility).** Given a  $\lambda a$ -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  $\lambda a$ -context  $C'[\ ]$  and subterms  $P_1, \dots, P_n \sqsupseteq P$  (for some  $n \geq 0$ ) such that Let  $R \equiv (\lambda x. P)Q$  be a redex.  $M = C[R]$  is  $R$ -admissible if

- A.  $P \equiv C'[P_1, \dots, P_n]$ ,
- B. For all the  $P_i$ ,  $x \notin \text{fv}(P_i)$ .
- C. There is no sharing between  $C'[\ ]$  and the rest of  $M$ , *i.e.*,

$$\text{addr}(C[\ ]) \cap \text{addr}(C'[\ ]) = \emptyset$$

$$(M \text{ wfa ensures } (\text{addr}(P_1) \cup \dots \cup \text{addr}(P_n)) \cap \text{addr}(C'[\ ]) = \emptyset).$$

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

Next the copy operation.

**3.4.10 Definitions (copying).** Define the following additional reductions.

A. *Copying* is the sharing closure of

$$M^a \rightarrow M \quad (\text{cp})$$

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

This notion of reduction is clearly well-defined thanks to Proposition 3.4.4 and Theorem 3.4.11. 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  $\xrightarrow{\text{cp}}$ -steps.

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

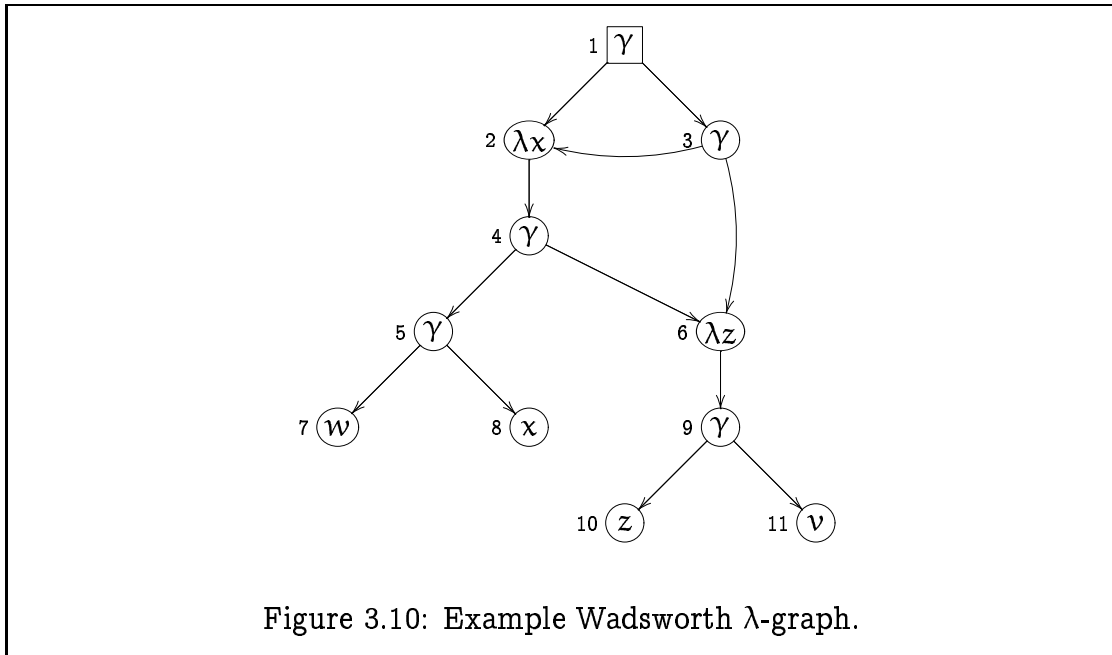
**3.4.11 Theorem.** *Given the  $\lambda$ -term  $M \equiv C[R]$  with  $R \equiv (\lambda x.P)Q$  and let  $M$  be  $R$ -admissible. Then  $C[R']$  with  $R' \equiv P[x := Q]$  is a  $\lambda$ -term.*

*Proof.* We know that  $R \equiv (\lambda x.P)Q$  with  $P \equiv C'[P_1, \dots, P_n]$  from Definition 3.4.9. The reduction must establish  $R' \equiv P[x := Q] \equiv C'[P_1, \dots, P_n][x := Q] \equiv C'[x := Q][P_1, \dots, P_n]$  since  $x \in \text{fv}(P_i)$ . 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.  $\square$

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

**3.4.12 History (Wadsworth's lambda-graphs).** The intuition of the above is to use 'wfa' to express that a preterm corresponds to a  $\lambda$ -term with properly defined sharing information. This follows the style of the  $\lambda$ -graph calculus of Wadsworth (1971, Chapter 4), where it is called *admissible* (more on this in History 3.4.13 below).

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



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

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

is depicted in this style in Figure 3.10 (mimicking Wadsworth 1971, Figure 5a) where <sup>a</sup> and <sup>b</sup> identify the nodes 2 and 6, respectively.

Furthermore notice how variables are syntactic in Wadsworth's graphs: instead of the variable convention Wadsworth explicitly requires all variables occurring in the graph to be unique, and Wadsworth allows 'constants' which we will not discuss here (but see the treatment of CRS in chapter 5). Except for this it can be seen that our  $\lambda$ -graph notion is very close to Wadsworth's.

**3.4.13 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  $\lambda$ -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  $\lambda$ -body copying, when  $C'[]$  is as large as possible (*i.e.*, is  $P$ ).

Finally we can define the shared  $\beta$ -reduction step using the techniques introduced above.

**3.4.14 Definitions ( $\lambda$ a-reduction).** Define the following reductions on  $\Lambda_a$  (modulo  $\equiv$ ).

- A. *R-contraction* is the sharing closure of the ( $\beta$ ) rule.
- B. *Shared  $\beta$ -reduction* of  $C[R]$ , notation  $\dashv\vdash_{\beta_a}$ , is defined as  $R$ -copying followed by  $R$ -contraction.

This notion of reduction is clearly well-defined thanks to Proposition 3.4.4 and Theorem 3.4.11. 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  $\dashv\vdash_{cp}$ -steps.

**3.4.15 History.** Wadsworth (1971, p.160) actually defines a  $\text{Copy}(R, G)$  operation concretely as doing complete  $\lambda$ -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  $\beta$ -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.

**3.4.16 Definition (sharing stable).** A reduction  $\rightarrow$  is *sharing stable* iff  $\xrightarrow{+} \subseteq \rightsquigarrow$ , *i.e.*, if the sharing extension is sound (the opposite of Proposition 3.4.6).

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

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

- A. The sharing extension of  $\xrightarrow{\text{cp}} \cdot \rightsquigarrow$  is sharing stable.
- B. The sharing extension of the relation  $\xrightarrow{\text{cp}} \cdot \rightsquigarrow$  restricted to contexts given by

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

is sharing stable.

*Proofs.* A is trivially true since the complete unraveling constructed by  $\xrightarrow{\text{cp}}$  is obviously admissible for any reduction. B is equally trivial since ‘unshared’ implies admissibility in any context.  $\square$

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.

### 3.4.18 Comparisons.

- A. 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  $\xrightarrow{+\beta_a}$  is explicitly not used by Felleisen and Friedman because their calculus would then perform a wrong substitution of a bound variable. The problem is described as “establishing the sharing relation too early” – the same thing happens with  $\xrightarrow{+\beta_a}$  if one insist on a strategy that breaks admissibility but we have the choice of not doing this.

- B. 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  $\hat{\wedge}$  operator used to enforce the uniqueness of addresses in substitution bodies.
- C. The acyclic component of  $\lambda$ -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  $\lambda\beta$ -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  $\lambda$ -abstraction. We have more to say for both cases when we treat cyclic data in section 5.2.

## 3.5 Explicit Substitution & Sharing

In this section we demonstrate how explicit sharing can be added to the  $\lambda\text{xgc}$ -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  $\lambda$ -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  $\lambda\text{a}$  and  $\lambda\text{xgc}$ , including the notion of reduction.

**3.5.1 Definition ( $\lambda\text{xa}$ -terms).** The set of  $\lambda\text{xa}$ -preterms is ranged over by the letters  $MN \dots$  (*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  $\lambda\text{a}$ -terms). The wfa and unraveling definitions are lifted from Definition 3.4.1; the notions of free variables, renaming, and  $\alpha$ -equivalence, are defined using the combination of the rules for  $\lambda\text{a}$ - and  $\lambda\text{x}$ -terms in the obvious way. As usual the  $\lambda\text{xa}$ -terms  $\Lambda\text{xa}$ , are the wfa preterms modulo  $\equiv$ .

So what should a sharing reduction step look like with explicit substitution? The obvious generalisation of  $\lambda$ xgc-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  $\lambda$ xgc-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  $\beta$ - or x-redex. Collecting this insight we get the following system, our starting point.

**3.5.2 Definition ( $\lambda$ xgca-reduction step).**  $\xrightarrow{\text{bxgca}}^{\text{cp}}$  is the contextual closure of

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

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

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 3.4.9 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  $\lambda$ xgca.

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

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

$$\text{addr}(C[]) \cap \text{addr}(C'[]) = \emptyset$$

( $M$  wfa ensures  $(\text{addr}(P_1) \cup \dots \cup \text{addr}(P_n)) \cap \text{addr}(C'[]) = \emptyset$ ). Furthermore, if  $M$  is  $R$ -admissible for all  $N \equiv C_R[\vec{P}] \triangleright M$  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  $\lambda\text{xgca}$ -calculus; notice that this, like Proposition 3.4.17, is the opposite of the (also here inherent) Proposition 3.4.6.

**3.5.4 Lemma.** If  $\dashv\vdash$  is locally sharing stable for  $\lambda\text{xgca}$ -reduction then it satisfies

$$\text{for any } \lambda\text{x-term } M, \begin{array}{ccc} \square & \dashv\vdash^+ & \square \\ \text{cp} \downarrow \text{---} & & \downarrow \text{cp} \\ M & \xrightarrow{\text{bxgc}} & \blacksquare \end{array} .$$

*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  $\lambda\text{xgc}$ -redex because any  $\lambda\text{xgc}$ -redex will be reestablishable as a corresponding  $\lambda\text{xgca}$ -redex.  $\square$

**3.5.5 Theorem.**  $\xrightarrow[\text{cp}]{} \dashv\vdash$  is a projection from  $\xrightarrow[\text{bxgca}]{}^+$  to  $\xrightarrow[\text{bxgc}]{}^+$

*Proof.* One way is Proposition 3.4.6, the other Lemma 3.5.4.  $\square$

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.

**3.5.6 Corollary.**  $\xrightarrow[\text{bxgca}]{}^+$  is a conservative extension that preserves strong normalisation of  $\xrightarrow[\beta]{}^+$ .

*Proof.* Combination of Theorem 3.5.5 and Theorem 3.1.17.  $\square$

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  $\lambda_{\text{let}}$ -terms are define by

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

where  $xyz$  range over variables.

**Axioms.** Reduction,  $\xrightarrow{\text{need}}$ , is the contextual closure of

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

Figure 3.11: The Call-by-Need calculus,  $\lambda_{\text{let}}$ .

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 substitutions – copying – underneath that point. This is what took so much effort with directly abstractable subterms/maximal free subexpressions in the general substitution case.

**3.5.7 Comparison (Call-by-Need).** An interesting comparison is with the ‘standard reduction’ of  $\lambda_{\text{let}}$ -calculus of Ariola, Felleisen, Maraist, Odersky and Wadler (1995, Figure 2), reproduced in Figure 3.11. This implements the *Call-by-Need* reduction strategy which is weak reduction with sharing of unevaluated terms.

This is close to  $\lambda x_{\text{gc}}$  with a *Call-by-Value restriction on copying* similarly to what we would get by restricting (cpx) to

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

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  $\lambda$ -bound variables in  $\lambda_{\text{let}}$  (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 5. And in fact the problem is also present implicitly in  $\lambda_{\text{let}}$  through the fact that the rule (let-V) allows ‘instant substitution’ into any context.

Another interesting aspect of  $\lambda_{\text{let}}$  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 3.2.

**3.5.8 Discussion (complexity).** We have argued throughout that combination of explicit substitution and sharing gives a representation of the  $\lambda$ -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 consequently *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 ‘shuffling’ 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 3.3.

**3.5.9 Discussion (sharing vs de Bruijn indices).** As discussed in section 2.5, the original definition of de Bruijn (1972) used three concepts for the implementation of substitution (shown in Definition 2.5.1), and the various naming principles can be 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 3.5 where the two  $\lambda\nu$ -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.

## 3.6 Summary

This chapter has studied many of the directions one can obtain by combining the new *explicitness dimensions* classification.

The introduced explicit substitution calculus,  $\lambda x$ , is new. The PSN of  $\lambda\beta$  result specific to  $\lambda xgc$ , proven in section 3.2, is new (joint work with Roel Bloo), in particular no direct proof was known before (we attribute this to the use of garbage collection, also a new technique).

The categorisation through translations of section 3.3 is new, as is the notion of ‘strict projection’ to generalise the PSN result of  $\lambda x$  to the other calculi; this gives a family of direct proofs of PSN results where only indirect ones were known before.

The approach to general sharing introduced first in section 3.4 is a simplification of the ‘folklore’ of parallel reduction, and contributes by making clearer the notion of sharing for terms. This is exemplified by the much simplified presentation of sharing  $\lambda$ -graph reduction derived from it and Wadsworth’s (1971) calculus.

The combination of explicit substitution and sharing into a calculus modeling 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 1995, 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  $\lambda$ -calculus implementations – an important first step towards solving the problem for functional languages. As a side contribution we observe that  $\lambda xa$  is  $\beta$ -PSN which guarantees that our quest for ‘realistic’ computation times has not introduced the possibility of infinite reductions.



# 4

## Machines for $\lambda$ -calculus

The goal of this chapter is to extend the operational insights of the previous chapter ‘all the way’ to *abstract machines* that behave like concrete machines (*aka* ‘real computers’) with respect to some observable aspects, typically computation time, whereas they (still) abstract away other aspects, typically the encoding of programs and data as strings of bits.

It is well known since Landin (1964) how one can obtain from a *deterministic and compositional strategy* (for  $\lambda$ -calculus) a ‘machine’ that performs *mechanical evaluation* (of  $\lambda$ -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 chapter are maintained in abstract machines*. This extends the usefulness of abstract 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 order to make the machines ‘mechanic’, *i.e.*, deterministic, we still couple with the reduction strategies of section 2.4. However, compared to traditional approaches we have thus swapped the decision of the strategy with the other implementation decisions. This gives way for two things: first it makes it possible to discuss the ‘overhead’ of explicit aspects; this is practical when arguing the case that a particular aspect should be ignored. Second it makes it possible to see which strategy/explicitness combinations are particularly simple in the sense that the complicated considerations required when considering some of the explicitness dimensions disappear, or rather, collapse into simple ones. This gives a qualitative understanding of why most published abstract machines are much simpler than one should expect given the calculi they claim to implement by exhibiting the aspects that ‘collapse’ due to the restriction to weak closed terms.

We commence by demonstrating how the  $\lambda$ x-calculus can be used as the basis for deriving abstract machines for two common reduction strategies: CBN in section 4.2 and CBV in section 4.3. Section 4.4 shows how they generalise to a graph-reduction-like machines when *explicit sharing* is added. Throughout these developments we insist on using *names* just as the  $\lambda$ x-calculus they are based on.

## 4.1 Abstract Machines

By *abstract machine* we generally denote reduction systems where each reduction step can be implemented as a ‘simple’ operation on a concrete machine (by ‘simple’ we mean that it can be executed in constant time on a computer). This section defines this for our purpose and explains our method for *systematic derivation* of such machines. We will pay special attention to how one ensures that the *reduction step count* of such machines is a faithful *complexity measure* for  $\lambda$ -reduction.

We have attempted to stay within the tradition of Landin’s (1964) SECD machine and Plotkin’s (1981) *transition systems* but have also been inspired by Hannan and Miller (1990) and the classical book of Henderson (1980).

We first define what an abstract machine is, then we outline the *derivation principle*, and finally we will discuss complexity issues related to abstract machines.

Our definition of an abstract machine is based on the insights of the previous chapter when considering what is efficiently implementable on a computer. This involves the *locality* issue that we have already discussed in the previous chapter: we need to couple the already described explicit substitution and explicit sharing with the new notion of *explicit strategy*, as usual we attempt to define it as generally as possible. The only other abstract definition we know of is that of Lescanne (1994b) which is contained in this one.

#### 4.1.1 Definitions (abstract machine).

- A.  $\rightarrow$  is *local* if there is a bound  $k \in \mathbb{N}$  such that for every possible rewrite step  $t_1 \rightarrow t_2$  there exists terms and bounded contexts such that
  - $t_1 = C[C_1[\vec{s}_{(n)}]]$ .
  - $t_2 = C[C_2[\vec{t}_{(m)}]]$ , where  $\forall j \in \{1, \dots, m\} \exists i \in \{1, \dots, n\} : s_i = t_j$ , and
  - $k > |C_1[[]]|$  and  $k > |C_2[[]]|$  (*i.e.*, the contexts are bounded).
- B.  $\rightsquigarrow$  is *primitive* if it is a function and there exists some local and strongly normalising reduction  $\rightarrow$  such that  $\rightsquigarrow \subseteq \rightarrow$ .
- C. A reduction is an *abstract machine implementing*  $\rightarrow$  if it is a local and flat reduction system  $\rightsquigarrow$  with a primitive projection from  $\rightsquigarrow$  to  $\rightarrow$ .
- D.  $\rightarrow$  is *space local* if it is local as above and furthermore
  - the  $t_i$  are pairwise distinct.

If an abstract machine is space local then we say it is *space faithful*.

The intuition behind these restrictions is as follows: the rules should be *local* because they should only affect a bounded amount of the machine memory. They should be *flat* because then a redex can be found immediately since it must at the root of the term. And there should be a *primitive* translation from the relation it implements in order to avoid tricks being played with using excessively expressive translations to do all the work. A space local system should furthermore not duplicate any subterms of unknown size.

Plotkin (1981) calls an abstract machine like this for a *transition system* and the rules<sup>1</sup> for *transitions*. Traditionally terms of such systems are called for [the machine's] *configurations* or sometimes even *states*. We will use all these words interchangeably below. Abstract machines also traditionally have *instructions* which means that terms must be 'compiled' into a linear form before the machine can execute them – this constraint is no longer common and we will not talk further about it (except when comparing with related work).

The above definition is carefully formulated to ensure that our considerations about complexity are well founded. It would be no good to have two realistically implementable reduction systems related by a projection which is not computable – at least it would not teach us much about the relative complexities of the two! This is why we introduce the notion of *primitive* above: to ensure that the transition from one system to another is not too complicated: a primitive relation is always strongly normalising even when using local steps. We will make use of this.

**4.1.2 Proposition (substitution is primitive).**  $\xrightarrow{x}$  is primitive on  $\Lambda x$ .

*Proof.*  $\xrightarrow{x}$  is a total function on  $\Lambda x$  (with range  $\Lambda$ ),  $\xrightarrow{x} \subseteq \xrightarrow{x}$ ,  $\xrightarrow{x} \rightarrow \text{SN}$ , and finally it is not difficult to see from Definition 3.1.4.B that  $\xrightarrow{x}$  is local.  $\square$

We already know how to obtain a local system through explicit substitution of section 3.1, and even a space local one by augmenting with explicit sharing of section 3.5. However, we still need a methodology for finding a flat system. This is what we will do below, essentially encoding strategies in the rewriting, inspired by the techniques of Hannan and Miller (1990, we return to these in Comparison 4.2.16.C).

**4.1.3 Principle (abstract machine derivation).** Start with a compositional and local reduction system; we will only consider deterministic ones. Then go through successive *transformations* of the rewrite system, where each transformation consists of the following phases:

**Search for Theorem:** prove a *structural property* of the system that might be useful in an implementation. Typically one of the inferences or a redundancy.

**Represent:** find a *term structure representation* that makes it possible to exploit the proved property explicit *inside* the system, encoding it (and hence flattening the terms).

---

<sup>1</sup>Often the rules and the set of rewrites is confused.

**Fold:** transform the system to *exploit the property* by distinguishing between forms of the term such that the property is represented in a stable manner.

**Minimalise:** simplify the system to remove redundant rules, deterministic chains of rewrites, *etc.*

The process is finished when the result is an abstract machine.

This is of course most useful if each transformation is so mechanic that correctness is immediate; ultimately the goal is to mechanise the procedure completely.

## 4.2 A Call-by-Name Abstract Machine

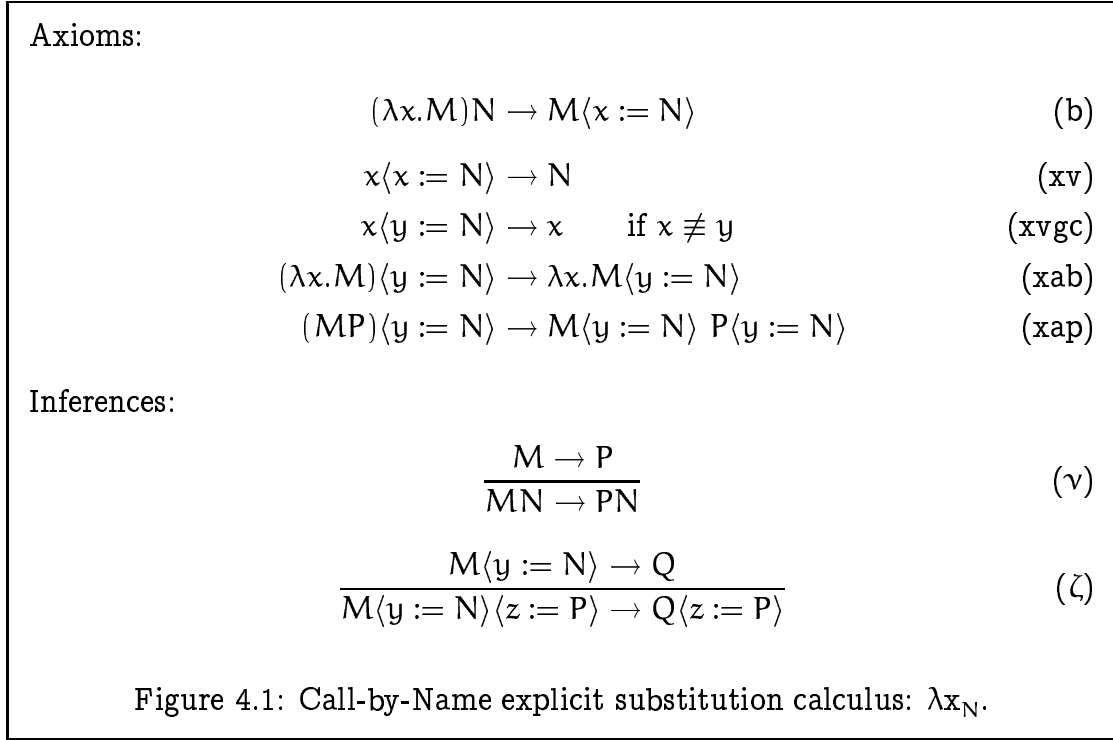
In this section we demonstrate how the  $\lambda x$ -calculus of section 3.1 can be used as the basis for deriving an abstract machines for one particular strategy: Plotkin's (1975) *Call-by-Name* (CBN) strategy of Definition 2.4.5.B. The derivation looks similar to other such derivations, however, we attempt to do it rigorously, following Principle 4.1.3, such that it is easy to extend the derivation to include further explicitness dimensions as we shall do in section 4.4.

Thus our starting point is the calculus obtained by restricting the  $\lambda x$ -reduction relation  $\xrightarrow{\text{bx}}$  of Definition 3.1.4 to the strategy of  $\xrightarrow{\text{N}}$ . This involves our first design decision: how should the strategy be generalised to explicit substitution terms of the form  $M\langle x := N \rangle$ ? The most obvious way is to just replace  $(\beta)$  with a rule like

$$\frac{M\langle x := N \rangle \xrightarrow{x} Q}{(\lambda x.M)N \rightarrow Q}$$

and impose a suitable reduction strategy *separately* on  $\xrightarrow{x}$ . This is not very efficient, however, since it means that the resulting implementation will perform all substitutions regardless of whether they are needed – if used it will lead to what is known as a *substitution-based evaluation* (Mosses 1979) which is the traditional alternative to *environment-based evaluation* which is what we will use here and what all abstract machines use (Henderson 1980).

A more efficient method is to delay substitution as much as possible, and then the challenge is to delay reduction as much as possible whilst retaining in the new reduction exactly the normal forms of  $\xrightarrow{\text{N}}$  (the whnf). This is not so difficult to ensure: we take the reduction rules of  $\lambda x$ , add the strategy rules of CBN, and *complete* the system by considering all possible reducts of terms that



can be produced during a CBN reduction with  $\xrightarrow{N}$ . We then *add rules such that all non-normal forms reduce correctly*. Since we will adopt the usual *restriction to closed terms*,  $\mu_N$  is not applicable so the only context where this is problematic is for terms of shape  $M\langle y := N \rangle\langle z := P \rangle$ . We choose that they reduce by performing the substitutions, inventing the new context rule ( $\zeta$ ). We will resolve needed substitutions from the left, in accordance with our chosen delay principle.

**4.2.1 Definition ( $\lambda x_N$ -reduction).**  $\xrightarrow{\text{bx}_N}$  is the restriction of  $\lambda x$  to the CBN strategy, shown in Figure 4.1.

**4.2.2 Lemma ( $\lambda x_N$ , correctness).**  $\xrightarrow{\text{bx}_N} \gg$  is local and deterministic, and  $\xrightarrow{x} \gg$  is a primitive projection from it to  $\xrightarrow{N} \gg$ .

*Proof.* Locality is obvious from the rules: All the axioms are local and the inferences do not construct anything. The system is deterministic in a very obvious sense: for every term which is closed there is at most one applicable rule. In fact, for any  $\lambda x$ -term not in whnf there is *exactly* one applicable rule, as is easily verified. This also establishes the soundness of the projection; its



$(M, N_1 \cdot \dots \cdot N_k)$  and only reduce on  $M$ : then we get the same possible reductions until  $M$  becomes an abstraction. We use it by adding the representation as an axiom and fold the system. The resulting first machine approximation is the following system which we call  $\lambda x_N S$ .

**4.2.5 Definition (first intermediate  $\lambda x_N$ -machine:  $\lambda x_N S$ ).**  $\xrightarrow{\text{bx}_N S}$  is given by the axioms

$$\begin{aligned}
(MN, S) &\rightarrow (M, N \cdot S) && (S) \\
(\lambda x.M, N \cdot S) &\rightarrow (M\langle x := N \rangle, S) && (\text{bS}) \\
(x\langle x := N \rangle, S) &\rightarrow (N, S) && (\text{xvS}) \\
(x\langle y := N \rangle, S) &\rightarrow (x, S) \quad \text{if } x \neq y && (\text{xvgsS}) \\
((\lambda x.M)\langle y := N \rangle, S) &\rightarrow (\lambda x.M\langle y := N \rangle, S) && (\text{xabS}) \\
((M_1 M_2)\langle y := N \rangle, S) &\rightarrow (M_1\langle y := N \rangle, M_2\langle y := N \rangle \cdot S) && (\text{xapS})
\end{aligned}$$

and the inference

$$\frac{(M\langle y := N \rangle, S) \rightarrow (Q_1, Q_2 \cdots Q_n \cdot S)}{(M\langle y := N \rangle\langle z := P \rangle, S) \rightarrow ((Q_1 Q_2 \cdots Q_n)\langle z := P \rangle, S)} \quad (\zeta S)$$

As might be expected, folding of the rules with the representation has removed  $(\nu)$  completely since it would become an identity rewrite. However, this system is surprising with respect to one thing: if the reduct  $Q$  of  $(\zeta)$  is an application  $Q_1 Q_2 \cdots Q_n$  then most of it will have been folded onto the stack (as we call it) as shown.

The correctness is immediate.

**4.2.6 Lemma.** For closed  $M$ ,

$$M \xrightarrow{\text{bx}_N} \lambda x.N$$

if and only if

$$(M, \epsilon) \xrightarrow{\text{bx}_N S} (\lambda x.N, \epsilon)$$

The system is still not flat because of  $(\zeta S)$ . This is easily obtained, however, by using the following similarly simple observation based on  $(\zeta S)$ , where we for the first time cannot assume the terms are closed because substitutions *are* clearly reduced in a substitution context.



**4.2.7 Proposition.** For all  $M, N \in \Lambda_{\mathbf{x}}$  and  $S \in \Lambda_{\mathbf{x}}^*$ ,

$$(M\langle x := N \rangle, S) \xrightarrow{\text{bx}_N S} (Q_1, Q_2 \cdots Q_n \cdot S)$$

if and only if, for all  $P \in \Lambda_{\mathbf{x}}$ ,

$$(M\langle x := N \rangle\langle y := P \rangle, S) \xrightarrow{\text{bx}_N S} ((Q_1 Q_2 \cdots Q_n)\langle y := P \rangle, S)$$

The result is as follows, again correctness is immediate.

**4.2.8 Definition (second intermediate  $\lambda_{\mathbf{x}_N}$ -machine:  $\lambda_{\mathbf{x}_N} \mathbf{SE}$ ).**  $\xrightarrow{\text{bx}_N \mathbf{SE}}$  is given by the axioms

$$\begin{aligned} (M\langle x := N \rangle\langle y := P \rangle, E, S) &\rightarrow (M\langle x := N \rangle, \langle y := P \rangle E, S) && \text{(E)} \\ (MN, \epsilon, S) &\rightarrow (M, \epsilon, N \cdot S) && \text{(SE)} \\ (\lambda x.M, \epsilon, N \cdot S) &\rightarrow (M, \langle x := N \rangle, S) && \text{(bSE)} \\ (x\langle x := N \rangle, E, S) &\rightarrow (N, E, S) && \text{(xvSE)} \\ (x\langle y := N \rangle, E, S) &\rightarrow (x, E, S) \quad \text{if } x \neq y && \text{(xvgsE)} \\ ((\lambda x.M)\langle y := N \rangle, E, S) &\rightarrow (\lambda x.M\langle y := N \rangle E, \epsilon, S) && \text{(xabSE)} \\ ((M_1 M_2)\langle y := N \rangle, E, S) &\rightarrow (M_1, \langle y := N \rangle, M_2\langle y := N \rangle \cdot S) && \text{(xapSE)} \end{aligned}$$

**4.2.9 Lemma.** For closed  $M$ ,

$$(M, \epsilon) \xrightarrow{\text{bx}_N S^{\#}} (\lambda x.N, \epsilon)$$

if and only if

$$(M, \epsilon, \epsilon) \xrightarrow{\text{bx}_N \mathbf{SE}^{\#}} (\lambda x.N, \epsilon, \epsilon)$$

Again we transform the system by adding an axiom corresponding to the representation change  $(M\langle x := N \rangle, S)$  to  $(M, \langle x := N \rangle, S)$  and fold, noting that the non-substitution rules cannot be invoked in a substitution-context (because Proposition 4.2.7 only holds in a context with at least one substitution). This finishes the last derivation step.

**4.2.10 Definition ( $\lambda_{\mathbf{x}_N}$ -machine:  $\lambda_{\mathbf{x}_N} \mathbf{M}$ ).** The  $\lambda_{\mathbf{x}_N}$ -machine, defining  $\xrightarrow{\text{bx}_N \mathbf{M}^{\#}}$ , is shown in Figure 4.2.

$$\begin{array}{ll}
(M \langle x := N \rangle, E, S) \rightarrow (M, \langle x := N \rangle E, S) & \text{(EM)} \\
(MN, E, S) \rightarrow (M, E, NE \cdot S) & \text{(SM)} \\
(\lambda x.M, \epsilon, N \cdot S) \rightarrow (M, \langle x := N \rangle, S) & \text{(bM)} \\
(x, \langle x := N \rangle E, S) \rightarrow (N, E, S) & \text{(xvM)} \\
(x, \langle y := N \rangle E, S) \rightarrow (x, E, S) \quad \text{if } x \neq y & \text{(xvgcM)} \\
(\lambda x.M, \langle y := N \rangle E, S) \rightarrow (\lambda x.M \langle y := N \rangle E, \epsilon, S) & \text{(xabM)} \\
(M_1 M_2, \langle y := N \rangle E, S) \rightarrow (M_1, \langle y := N \rangle E, M_2 \langle y := N \rangle E \cdot S) & \text{(xapM)}
\end{array}$$

Figure 4.2: Abstract  $\lambda_{x_N}$ -machine:  $\lambda_{x_N}M$ .

**4.2.11 Lemma.** For closed  $M$  and  $\lambda x.N$ ,

$$(M, \epsilon, \epsilon) \xrightarrow{\text{bx}_N \text{SE}} (\lambda x.N, \epsilon, \epsilon)$$

if and only if

$$(M, \epsilon, \epsilon) \xrightarrow{\text{bx}_N M} (\lambda x.N, \epsilon, \epsilon)$$

This completes the formal bricks that we need.

**4.2.12 Lemma.** For closed  $\lambda$ -terms  $M$ ,

$$M \xrightarrow{N} \lambda x.P$$

if and only if

$$(M, \epsilon, \epsilon) \xrightarrow{\text{bx}_N M} (\lambda x.N, \epsilon, \epsilon) \text{ and } \downarrow_x(N) \equiv P$$

*Proof.* Follows from Lemma 4.2.6, Lemma 4.2.9, and Lemma 4.2.11.  $\square$

**4.2.13 Theorem ( $\lambda_{x_N}M$ , correctness).**  $\xrightarrow{\text{bx}_N M}$  is an abstract machine for  $\xrightarrow{N}$ .

*Proof.* We already have locality and the primitive projection from Lemma 4.2.2 and Lemma 4.2.12. Flatness is obvious.  $\square$

However, we cannot really be sure that the machine does the same number of reductions as the  $\lambda_{x_N}$ -system since we have allowed duplication of subterms – the machine is not space faithful. This is the motivation for incorporating sharing in section 4.4.

**4.2.14 Remark (variable convention).** Note there are still side conditions on variables, namely two implicit side conditions to  $(xabM)$ . We can eliminate these: the first,  $x \neq y$  (to avoid ‘variable clash’), is easy to check explicitly, and we can replace  $(xabM)$  with

$$(\lambda x.M, \langle x := N \rangle E, S) \rightarrow (\lambda x.M, E, S) \quad (xabM1)$$

$$(\lambda x.M, \langle y := N \rangle E, S) \rightarrow (\lambda x.M \langle y := N \rangle, E, S) \quad (xabM2)$$

since  $x \notin \text{fv}(\lambda x.M)$ . The second condition,  $x \notin \text{fv}(N)$  (to avoid ‘variable capture’), is trivially true because of the following:

**4.2.15 Proposition.** Assume  $M$  pure and closed and  $(M, \epsilon, \epsilon) \xrightarrow{\text{bx}_N \text{SE}} (N, \langle x_1 := N_1 \rangle \cdots \langle x_n := N_n \rangle, P_1 \cdots P_m)$ . Then all of  $N \langle x_1 := N_1 \rangle \cdots \langle x_n := N_n \rangle$ ,  $N_i$ , and  $P_i$ , are closed.

*Proof.* Easy case analysis of the rules that all preserve the property.  $\square$

**4.2.16 Comparisons (Call-by-Name machines).** Many published accounts of abstract machines and their close relatives, (non-flat) deterministic operational semantic inference systems, exist. Here we compare to a few that we have found significant, in chronological order.

- A. Henderson (1980) defines a *lazy SECD* machine which is realised by extending the CBV SECD machine (*cf.* Comparison 4.3.5.A) with **delay** and **force** primitive instructions that build *suspensions (aka thunks)* that correspond to closure of no arguments and serve as containers for inactive environments.
- B. Crégut (1990) presents two variations of an earlier (unpublished) abstract machine of Krivine which is known as the *Krivine machine*. This development is interesting because it includes correctness proofs of the exhibited machines based on the  $\lambda\sigma$ -calculus of Definition 3.3.21, and because the machines are so simple: the first is

$$\begin{aligned} ((C_1, C_2), e, p) &\rightarrow (C_1, e, \langle C_2 : e \rangle :: p) \\ (\lambda(C), e, f :: p) &\rightarrow (C, f :: e, p) \\ (0, \langle C : e_2 \rangle :: e_1, p) &\rightarrow (C_2, e_2, p) \\ (n + 1, f :: e, p) &\rightarrow (n, e, p) \end{aligned}$$

(where  $\langle \_ : \_ \rangle$  and  $\_ :: \_$  are binary constructions). Otherwise Crégut is mostly interested in evaluation to *full normal form* which is established by adding inference rules to the machine (the reverse of what we did above!) and then give a rather complex argument that this is correct.

- C. In the same volume one can find the exposition of Hannan and Miller (1990) where the normal order Krivine machine is derived using techniques very close to what we have presented. The difference is that Hannan and Miller (like Crégut) use de Bruijn indices for the derivation, which ‘freezes’ the stack discipline.
- D. Finally we mention Yoshida (1993) who presents a calculus of explicit substitution with names, much like  $\lambda x$ , which comes equipped with a (somewhat complicated) strategy that allows all sorts of reductions as long as they do not duplicate a redex. Yoshida is able to show that the chosen strategy gives the minimal number of what we would call ‘weak  $\lambda x$ -reduction steps’ which is a very nice result that was not believe true, however, the strategy is somewhat complicated and remains of mainly academic interest.

### 4.3 Towards a Call-by-Value Abstract Machine

In this section we show how an abstract machine can be derived from the  $\lambda x$ -calculus equipped with Plotkin’s (1975) *Call-by-Value* (CBV) strategy of Definition 2.4.5.A. The development parallels the one of the previous section hence we restrict ourselves to the highlights where the development is substantially different, in particular we do not formulate all the intermediate Lemmas.

Here our starting point is the calculus obtained by restricting the  $\lambda x$ -reduction relation  $\xrightarrow{\text{bx}}$  of Definition 3.1.4 to the strategy of  $\xrightarrow{\text{V}}$ . Again we decide to delay substitution as much as possible, obtaining the following initial system which is a combination of the  $\lambda x$ -reduction step and the restrictions of the CBV strategy on closed terms.

Consider the restriction of  $\lambda x$  of Definition 3.1.4,

$$\begin{aligned}
 (\lambda x.M)N &\rightarrow M\langle x := N \rangle && \text{if } N \text{ value} && (b_V) \\
 x\langle x := N \rangle &\rightarrow N && && (x_V) \\
 x\langle y := N \rangle &\rightarrow x && \text{if } x \neq y && (x_VGC) \\
 (\lambda x.M)\langle y := N \rangle &\rightarrow \lambda x.M\langle y := N \rangle && && (x_{ab}) \\
 (MP)\langle y := N \rangle &\rightarrow M\langle y := N \rangle P\langle y := N \rangle && && (x_{ap})
 \end{aligned}$$

to the strategy (inferences)

$$\begin{aligned}
 &\frac{M \rightarrow P}{MN \rightarrow PN} && && (\nu) \\
 &\frac{N \rightarrow P}{MN \rightarrow MP} && \text{if } M \text{ value} && (\mu_V) \\
 &\frac{M\langle y := N \rangle \rightarrow Q}{M\langle y := N \rangle\langle z := P \rangle \rightarrow Q\langle z := P \rangle} && && (\zeta)
 \end{aligned}$$

We can exploit that only closed terms are reduced to ‘fill in’ all possibilities for values and non-values to get rid of the side conditions: the three inference rules are mutually exclusive in an indirect way. Consider a term  $MN$ : since it is impossible for a value to reduce,  $(\nu)$  can never be applied when  $M$  is a value. Similarly for  $N$  and  $(\mu_V)$ , hence the seeming overlap between  $(b_V, \nu, \mu_V)$  is naught. Furthermore, neither of the rules can ever be applied when the subterm in question is a variable because that would require that reduction of applications was permitted under a binder which is not the case. These considerations have been exploited in the following,

**4.3.1 Definition ( $\lambda x_V$ -reduction).**  $\xrightarrow{bx_V}$  is the restriction of  $\lambda x$  to the CBV strategy, shown in Figure 4.3.

These rules are slightly more complicated structurally, hence we get a slightly longer derivation. The first ‘theorem’ we will exploit is  $(\nu')$ :

**4.3.2 Proposition.** For all  $M_1, M_2, P \in \Lambda x^\circ$ ,

$$M_1 M_2 \xrightarrow{bx_V} P$$

if and only if, for all  $N \in \Lambda x^\circ$ ,

$$M_1 M_2 N \xrightarrow{bx_V} PN$$

This results in introduction of a stack just as in the previous section, resulting in the following intermediate machine.

Axioms:

$$(\lambda x.M)(\lambda y.N) \rightarrow M\langle x := \lambda y.N \rangle \quad (\text{b}'_V)$$

$$x\langle x := N \rangle \rightarrow N \quad (\text{xv})$$

$$x\langle y := N \rangle \rightarrow x \quad \text{if } x \neq y \quad (\text{xvgc})$$

$$(\lambda x.M)\langle y := N \rangle \rightarrow \lambda x.M\langle y := N \rangle \quad (\text{xab})$$

$$(MP)\langle y := N \rangle \rightarrow M\langle y := N \rangle P\langle y := N \rangle \quad (\text{xap})$$

Strategy inferences:

$$\frac{M_1 M_2 \rightarrow P}{M_1 M_2 N \rightarrow PN} \quad (\text{v}')$$

$$\frac{N_1 N_2 \rightarrow P}{(\lambda x.M)(N_1 N_2) \rightarrow (\lambda x.M)P} \quad (\mu'_V)$$

$$\frac{M\langle y := N \rangle \rightarrow Q}{M\langle y := N \rangle\langle z := P \rangle \rightarrow Q\langle z := P \rangle} \quad (\zeta)$$

Figure 4.3: Call-by-Value explicit substitution calculus:  $\lambda x_V$ .

**4.3.3 Definition (first intermediate  $\lambda_{x_v}$ -machine:  $\lambda_{x_v}S$ ).**  $\xrightarrow{\text{bx}_v S}$  is given by the axioms

$$\begin{aligned}
(MN, S) &\rightarrow (M, N \cdot S) && (S) \\
(\lambda x.M, (\lambda y.N) \cdot S) &\rightarrow (M\langle x := \lambda y.N \rangle, S) && (b'_v S) \\
(x\langle x := N \rangle, S) &\rightarrow (N, S) && (xvS) \\
(x\langle y := N \rangle, S) &\rightarrow (x, S) \quad \text{if } x \neq y && (xvgsS) \\
((\lambda x.M)\langle y := N \rangle, S) &\rightarrow (\lambda x.M\langle y := N \rangle, S) && (xabS) \\
((M_1 M_2)\langle y := N \rangle, S) &\rightarrow (M_1\langle y := N \rangle, M_2\langle y := N \rangle \cdot S) && (xapS)
\end{aligned}$$

and the inferences

$$\begin{aligned}
&\frac{(N_1, N_2 \cdot S) \rightarrow (Q_1, Q_2 \cdots Q_n \cdot S)}{(\lambda x.M, (N_1 N_2) \cdot S) \rightarrow (\lambda x.M, (Q_1 Q_2 \cdots Q_n) \cdot S)} && (\mu'_v S) \\
&\frac{(M\langle y := N \rangle, S) \rightarrow (Q_1, Q_2 \cdots Q_n \cdot S)}{(M\langle y := N \rangle\langle z := P \rangle, S) \rightarrow ((Q_1 Q_2 \cdots Q_n)\langle z := P \rangle, S)} && (\zeta S)
\end{aligned}$$

So far this is as in the CBN case. However, now we observe that the new  $(\mu'_v S)$  rule *reduces something which is on the stack* – this was not the case in CBN and the basis for Proposition 4.2.7. This is not automatically resolved by the next step we have to take based on  $(\mu'_v S)$  as a property, again exploiting that this is never needed in a context under a binder. However, in order to avoid a very complicated representation we want to constrain the size of of the

The representation of this is slightly complicated by the fact that a group of terms is involved.

**4.3.4 Proposition.** For all  $\lambda x.M, N_1, N_2 \in \Lambda_{x^\circ}$  and  $S \in \Lambda_{x^*}$ ,

$$(N_1, N_2 \cdot S) \xrightarrow{\text{bx}_v S} (Q_1, Q_2 \cdots Q_n \cdot S)$$

if and only if, for all  $\lambda x.M \in \Lambda_{x^\circ}$ ,

$$(\lambda x.M, (N_1 N_2) \cdot S) \xrightarrow{\text{bx}_v S} (\lambda x.M, (Q_1 Q_2 \cdots Q_n) \cdot S)$$

The problem is to get in a compositional way the number  $n$  of  $Q$ s. This is routinely obtained, however, by pushing a *marker* onto the stack where the ‘.’ is, and a machine is obtained in one more step than the previous.

From this point on the development is completely similar to development of Hannan and Miller (1990); we will not delve into it but have shown it in Figure 4.4. The thing to notice is that we, as in the CBN case, get *closures for free*: since  $\beta$ -reduction only happens for closed terms, all binders are automatically pushed ‘inside’ the abstraction they belong to.

$(M\langle x := N \rangle, E, S) \rightarrow (M, \langle x := N \rangle E, S)$	( $E_V M$ )
$(MN, E, S) \rightarrow (M, E, NE \cdot S)$	( $S_V M$ )
$(\lambda x.M, \epsilon, (\lambda y.N) \cdot S) \rightarrow (M, \langle x := \lambda y.N \rangle, S)$	( $b_V abM$ )
$(\lambda x.M, \epsilon, (N_1 N_2) \cdot S) \rightarrow (N_1 N_2, \{\lambda x.M\}, S)$	( $b_V apM$ )
$(\lambda y.N, \epsilon, \{\lambda x.M\} \cdot S) \rightarrow (M, \langle x := \lambda y.N \rangle, S)$	( $b_V 'M$ )
$(x, \langle x := N \rangle E, S) \rightarrow (N, E, S)$	( $xv_V M$ )
$(x, \langle y := N \rangle E, S) \rightarrow (x, E, S) \quad \text{if } x \neq y$	( $xvgc_V M$ )
$(\lambda x.M, \langle y := N \rangle E, S) \rightarrow (\lambda x.M\langle y := N \rangle E, \epsilon, S)$	( $xab_V M$ )
$(M_1 M_2, \langle y := N \rangle E, S) \rightarrow (M_1, \langle y := N \rangle E, M_2\langle y := N \rangle E \cdot S)$	( $xap_V M$ )

Figure 4.4: Abstract  $\lambda_{x_V}$ -machine:  $\lambda_{x_V} M$ .

**4.3.5 Comparisons (Call-by-Value machines).** Almost as many published accounts exist of abstract machines for CBV with their own family of (non-flat) deterministic operational semantic inference systems; again we compare to a few significant ones.

- A. Landin's (1964) seminal paper includes a sketch of a small (tail) recursive ALGOL procedure for evaluating  $\lambda$ -expressions mechanically. The presentation most often used today is as a table due to Henderson (1980, section 6.2), where each of Landin's four parameters  $S$  (stack),  $E$  (environment),  $C$  (control), and  $D$  (dump), is a component of the *machine state* which is *transformed by instructions*, or more precisely: at any point in the evaluation, the instruction at the top of the  $C$  component is discarded and a suitable transformation takes place which ensures that a fresh instruction is available at the top of the  $C$  component again.. The latter is, perhaps, where the machine differs most strongly from our machines in that we interpret a  $\lambda$ -term directly as the 'instructions'.
- B. Hannan and Miller (1990) also apply their method to a CBV strategy and obtain, not surprisingly, a version of the SECD machine where the term is the 'instruction' which is very similar to ours except it uses de Bruijn indices instead of variables as do we (and the original SECD machine).



## 4.4 Abstract Machines with Sharing

In this section we demonstrate how explicit sharing and substitution as described in section 3.5 ripples through a generic abstract machine derivation for a CBN machine as presented in section 4.2, and shows up as essentially graph reduction. The essence in the exercise is to encode the *updating* function in the *strategy* because this will make it possible to describe locally what is updated when.

Since parallel reduction may involve reduction in any part of a term, the initial system is the following where the essential insight is that we *record any sharing points* and carefully use them to update whenever we have reduced.

**4.4.1 Definition ( $\lambda x a_N$ -reduction step).**  $\xrightarrow{\text{bxa}^a}$ , the ‘CBN  $\lambda x$ -reduction with updating of address  $a$ ’ is inductively defined in Figure 4.5. The  $a$  annotation on the arrow is called the ‘update address’;  $\epsilon$  is a unique *dummy address*. Notice that substitutions never set an update address.

As should be apparent, the strategy *always operates on an up-to-date term* since all parts of the context are updated by  $(\nu)$ . This is called an *update frame* by Peyton Jones (1992).

With this we discuss the derivation. The first folding is the combination of  $(\nu)$  and (Share) since those are the context-free rules. The folding proceeds very much like the use of Proposition 4.2.4 in section 4.2 but with the context needed for all three rules, thus the stack can contain both addresses and arguments.

We will not show this intermediate machine but instead go straight to the result of also folding  $(\zeta)$  into the system similarly to the use of Proposition 4.2.7. We only have to ascertain that the address inside each substitution is never removed by our strategy, then it is easy to see that all the rules fold without loss of sharing. This gives the machine in Figure 4.6 for which we have the following:

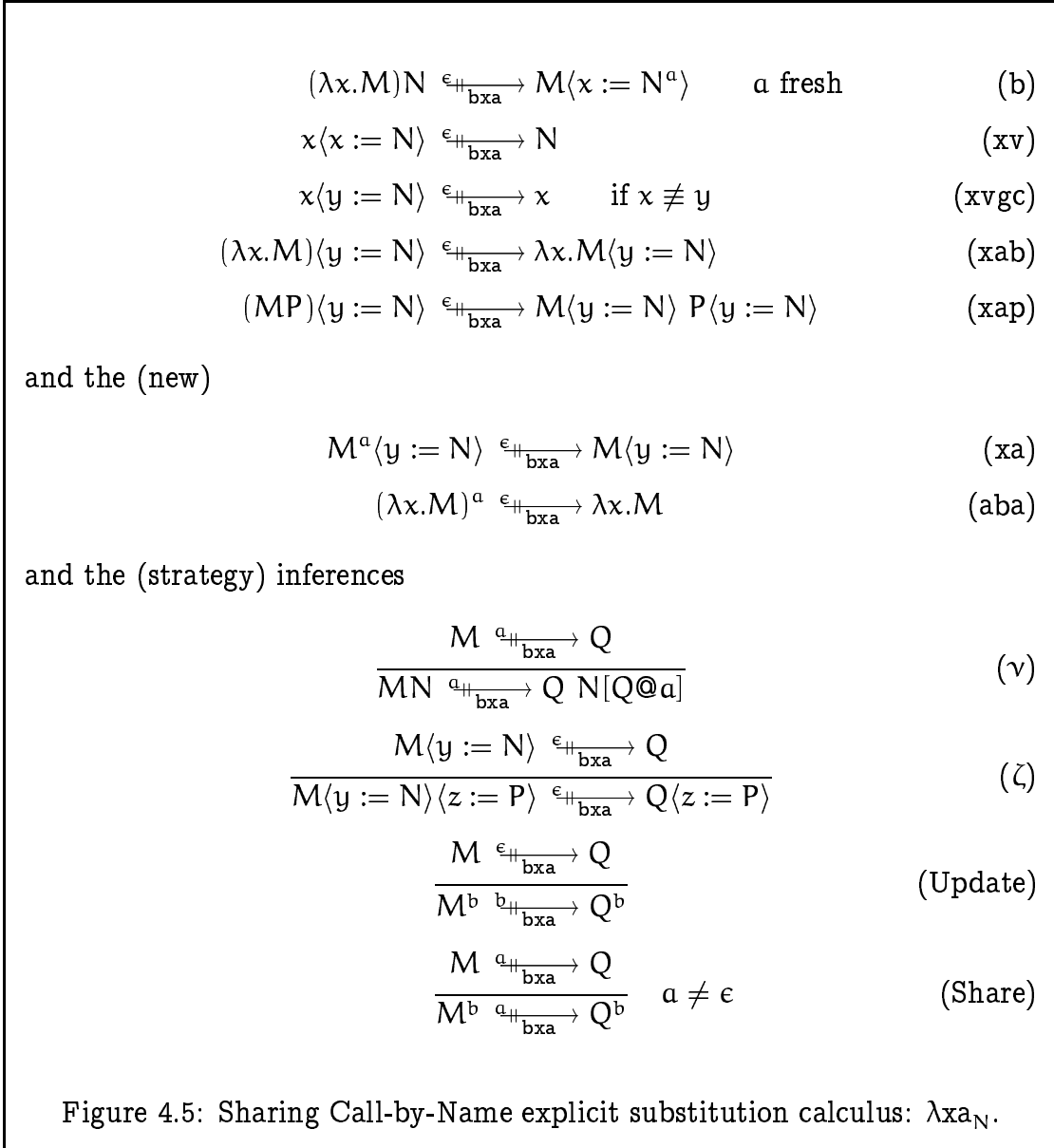
**4.4.2 Lemma.** For closed  $\lambda$ -terms  $M$ ,

$$M \xrightarrow{N} \lambda x.P$$

if and only if

$$(M, \epsilon, \epsilon) \xrightarrow{\text{bx}_N M} (\lambda x.N, \epsilon, \epsilon) \text{ and } \downarrow_x(N) \equiv P$$

*Proof.* Follows by a transformation correctness chain similar to that of section 4.2.  $\square$



$$\begin{array}{ll}
(M\langle x := N \rangle, E, S) \rightarrow (M, \langle x := N \rangle E, S) & \text{(E)} \\
(MN, E, S) \rightarrow (M, E, NE \cdot S) & \text{(M)} \\
(M^a, \epsilon, S) \rightarrow (M, \epsilon, a \cdot S) & \text{(SaE)} \\
(\lambda x.M, \epsilon, a \cdot S) \rightarrow (\lambda x.M, \epsilon, S[(\lambda x.M)^a]) & \text{(aM)} \\
(\lambda x.M, \epsilon, N \cdot S) \rightarrow (M, \langle x := N^a \rangle, S) \quad a \text{ fresh} & \text{(baM)} \\
(x, \langle x := N \rangle E, S) \rightarrow (N, E, S) & \text{(xvM)} \\
(x, \langle y := N \rangle E, S) \rightarrow (x, E, S) \quad \text{if } x \neq y & \text{(xvgcM)} \\
(\lambda x.M, \langle y := N \rangle E, S) \rightarrow (\lambda x.M\langle y := N \rangle E, \epsilon, S) & \text{(xabM)} \\
(M_1 M_2, \langle y := N \rangle E, S) \rightarrow (M, \langle y := N \rangle E, M_2 \langle y := N \rangle E \cdot S) & \text{(xapM)} \\
(M^a, \langle y := N \rangle E, S) \rightarrow (M, \langle y := N \rangle E, S) & \text{(xaM)}
\end{array}$$

Figure 4.6: Abstract  $\lambda x_N$ -machine with sharing:  $xa_N M$ .

Notice how the machine actually does the updating of the shared instances explicitly in (aM). This corresponds to the update that a ‘real’ machine would *already* have done at the time where the reduction happened, however, (aM) highlights that it is safe to *delay resynchronisation of the sharing* until a normal form is reached. Similarly, (xaM) highlights the fact that substituting into a term copies it (the  $a$  is removed). In short, insights about acyclic  $\lambda$ -graph reduction can be obtained from this system even though it was developed by theoretical means.

**4.4.3 Comparisons (lazy machines).** Only a few proper operational accounts exist of lazy reduction with sharing (we postpone discussion of cycles and recursion to the next chapter).

- A. The two classical abstract machines for lazy evaluation with sharing are, of course, the *G-machine* of Johnsson (1984) and its main derivative, the *STG-machine* (Peyton Jones and Salkild 1989, Peyton Jones 1992), and the derivatives of Fairbairn and Wray’s (1987) *Three Instruction Machine*, *TIM* (see also Peyton Jones 1987, Peyton Jones and Lester 1992, Plasmeijer and van Eekelen 1993). These all have in common that they are based on a low-level model of the computer memory, called *store* or *heap*, as a graph. The machine instructions then manipulate this graph by maintaining a stack of pointers into it (and the two schools basical-

ly differ in the way this is done: the G-machine keeps one stack whereas the TIM has the stack distributed in *environment frames* throughout the heap).

- B. Benaissa and Lescanne (1995) present an abstract machine framework, *triad machines* that handles lazy evaluation as well as other paradigms. The price of the generality in the machine, however, is that part of the specification is external to the formalism, notably the strategy, thus the triad approach cannot really be used for mechanical reasoning as we envision it. The triad machine is interesting with respect to one more issue: it includes garbage collection; it will be interesting to follow its development in the future.
- C. Three further works deserve special mention: Purushothaman and Seaman (1992), and Launchbury (1993), give operational semantic specifications for sharing very much like the one above except they, in fact, include also recursion in a way very similar to what we will get at in the next chapter, however, the systems are not derived in our sense. In contrast, Sestoft (1994) derives a lazy abstract machine much as we have done, and discusses in some depth the exact correspondence among these and the other machines mentioned above.

## 4.5 Summary

We have demonstrated how explicit substitution with names 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.

# 5

## Operational Combinatory Reduction Models

In this chapter we demonstrate how the techniques of the previous chapters can be applied to Klop's (1980) *combinatory reduction systems* (CRS, explained in section 2.6) 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 5.1, 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 5.2, we generalise *sharing* to CRSs and it turns out this combination will also permit special treatment of *explicit recursion*. Finally, we discuss in section 5.3 how the class realises operational aspects of an actual functional programming language, namely Plotkin's (1977) PCF with sharing.

## 5.1 Explicit Substitutes

In this section we show how the explicit substitution idea developed for the  $\lambda$ -calculus in section 3.1 can be generalised to CRS systems, and in particular we show that there is a translation from any CRS to an explicit substitution one.

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  $\lambda$ -calculus: to perform substitution in a stepwise manner, or, put differently, to change metareduces 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 3.1 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.

### 5.1.1 Example ( $\lambda\text{xgc}$ -reduction as CRS). $\lambda\text{xgc}$ is the CRS over terms

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

with  $\lambda\text{xgc}$ -CRS rules

$$\begin{aligned} (\lambda x.Z(x))Y &\rightarrow \Sigma([x]Z(x), Y) && \text{(b)} \\ \Sigma([x]x, Y) &\rightarrow Y && \text{(xv)} \\ \Sigma([x]Z, Y) &\rightarrow Z && \text{(xgc)} \\ \Sigma([x]\lambda y.Z(x, y), Y) &\rightarrow \lambda y.\Sigma([x]Z(x, y), Y) && \text{(xab)} \\ \Sigma([x](Z_1(x))(Z_2(x)), Y) &\rightarrow (\Sigma([x]Z_1(x), Y))(\Sigma([x]Z_2(x), Y)) && \text{(xap)} \end{aligned}$$

(as usual 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  $\lambda\text{xgc}$  in Definition 3.1.4: they are identical except for syntax conventions (for instance the  $\lambda\text{xgc}$ -term  $x(x := \lambda y.y)$  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 ( $\lambda\text{xgc}$  rule (xv)) and the more general garbage collection ( $\lambda\text{xgc}$  rule (gc)): both are special cases of the rule (xgc) above.

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

$$\begin{array}{ccc}
\boxed{\Sigma \left( \boxed{[x] \left( \boxed{(\lambda a. Z(a, x)) (Z_2(x))}, Y \right)} \right)} & \boxed{\Sigma([x] \lambda a. X(a), Y)} & \boxed{\Sigma([x] Z_1 Z_2, Y)} \\
\text{(xap)} & \text{(xab)} & \text{(xap)} \\
\text{(xap)} & \text{(xgc)} & \text{(xgc)}
\end{array}$$

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  $\lambda$ -calculus from section 3.1 is 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.

**5.1.2 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.<sup>1</sup> 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.

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

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

<sup>1</sup>This implies that the variables in  $\vec{x}$  are distinct.

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

- E.  $A([x, y]A(X(x, y))) \rightarrow A([x, y]B(X(x, y), X(y, x)))$  is *not* explicit because in the second we need to change one of the variables *inside* what matched  $X(x, y)$ .
- F.  $A([x, y]B(X(x, y), Y(x, y))) \rightarrow A([x, y]B(X(x, y), Y(y, x)))$  is *not* explicit for the same reason.
- G.  $A([x, y]B(X(x, y), X(y, x))) \rightarrow A([x, y]B(X(x, y)))$  is explicit.
- H.  $C([x]X(x), [y]X(y)) \rightarrow A([x]X(x))$  is explicit because we can just *choose* to use the bound variable that we know to be in the  $X$ -metaapplication we matched.
- I.  $A([x]X(x)) \rightarrow B([x, y]X(x), [y]X(y))$  is *not* explicit because in *one* of the  $X$ -constructions in the contractum we will have to substitute in a new variable.

#### 5.1.4 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.  $\square$

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  $\lambda$ -calculus, namely *substitution* as discussed in section 3.1 – 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.



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

**5.1.6 Examples.**  $(\beta)$  is saturated,  $(\eta)$  of Remark 2.3.9 is not, neither is the  $\lambda\text{xgc}$ -CRS shown in Example 5.1.1 because of  $(\text{xgc})$ .

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

$$\Sigma([x]Z^v, Y) \rightarrow Z$$

exploiting that  $Z^v$  cannot contain  $x$  (because then we should have written  $Z^v(x)$ ) 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 chapter 6. Using this rule instead of  $(\text{xgc})$  of course removes the two last overlaps.

**5.1.8 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  $\lambda$ -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 2.6.8) into the new rules.

**5.1.9 Definition (CRS explicification).** Given a CRS  $R$  with alphabet  $F_{\vec{i}}^n$ . The ESCRS  $R_x$  is obtained by the steps listed in Figure 5.1. If  $R$  defined the relation  $\rightarrow$  then we will denote the relation defined by  $R_x$  as  $\xrightarrow{\text{ie}}$ ; the subrelation consisting of only the introduction rules (with name  $(r-x)$ ) is denoted  $\xrightarrow{\vec{i}}$ , and the subrelation containing only the substitution distribution and elimination rules (with names of form  $(x-)$ ) is denoted  $\xrightarrow{e}$ ; we write  $\xrightarrow{\text{xgc}}$  for the union of all  $(\text{xgc-n-i})$  rules. Note that there are no rules for interaction between two substitutions.

**Restricted terms.** The restricted terms of the CRS  $R_x$  are defined by extending the syntax for  $R$  with the clause  $\Sigma^{n+1}([\vec{x}]t, \vec{t})$  where the symbols  $\Sigma^2, \dots$  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  $R_x$  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  $\Sigma^{n+1}$  symbol add a rule

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

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

$$\begin{aligned} \Sigma^{n+1}([\vec{x}]F^m(Z_1(\vec{x}), \dots, Z_m(\vec{x})), \vec{X}) & \quad (\text{x-F}^m\text{-n}) \\ \rightarrow F^m(\Sigma^{n+1}([\vec{x}]Z_1(\vec{x}), \vec{X}), \dots, \Sigma^{n+1}([\vec{x}]Z_m(\vec{x}), \vec{X})) & \end{aligned}$$

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

$$\begin{aligned} \Sigma^{n+1}([\vec{x}]x_i, \vec{X}) & \rightarrow X_i \quad (\text{xv-n-i}) \\ \Sigma^{n+1}([\vec{x}]Z(\vec{x}'), \vec{X}) & \rightarrow \begin{cases} Z & \text{if } n = 1 \\ \Sigma^n([\vec{x}']Z(\vec{x}'), \vec{X}') & \text{if } n > 1 \end{cases} \quad (\text{xgc-n-i}) \\ \text{where } \vec{x}' & = (x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) \\ \vec{X}' & = (X_1, \dots, X_{i-1}, X_{i+1}, \dots, X_n) \end{aligned}$$

(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 5.1: Explicification of CRS  $R$  into ESCRS  $R_x$ .

**5.1.10 Example.** The system  $\lambda\beta\mathbf{x}$  generated this way for the  $\lambda\beta$ -calculus with the rule

$$(\lambda\mathbf{x}.Z(\mathbf{x}))Y \rightarrow Z(Y) \quad (\beta)$$

is the following:<sup>2</sup>

$$\begin{aligned} (\lambda\mathbf{x}.Z(\mathbf{x}))Y &\rightarrow \Sigma([\mathbf{x}]Z(\mathbf{x}), Y) && (\beta\text{-}\mathbf{x}) \\ \Sigma([\mathbf{x}](Z_1(\mathbf{x}))(Z_2(\mathbf{x})), Y) &\rightarrow (\Sigma([\mathbf{x}]Z_1(\mathbf{x}), Y))(\Sigma([\mathbf{x}]Z_2(\mathbf{x}), Y)) && (\mathbf{x}\text{-}\textcircled{\@}) \\ \Sigma([\mathbf{x}]\lambda(Z(\mathbf{x})), Y) &\rightarrow \lambda(\Sigma([\mathbf{x}]Z(\mathbf{x}), Y)) && (\mathbf{x}\text{-}\lambda) \\ \Sigma([\mathbf{x}]\mathbf{x}, Y) &\rightarrow Y && (\mathbf{x}\mathbf{v}) \\ \Sigma([\mathbf{x}]Z, Y) &\rightarrow Z && (\mathbf{x}\text{gc}) \\ \Sigma([\mathbf{x}, \mathbf{y}]Z(\mathbf{x}, \mathbf{y}), Y) &\rightarrow [\mathbf{y}]\Sigma([\mathbf{x}]Z(\mathbf{x}, \mathbf{y}), Y) && (\mathbf{x}\text{ma}) \end{aligned}$$

It is the same as the  $\lambda\mathbf{x}\text{gc}$  CRS shown above except that the abstraction distribution rule ( $\mathbf{x}\text{ab}$ ) has been split into two steps: ( $\mathbf{x}\text{-}\lambda$ ) and ( $\mathbf{x}\text{ma}$ ). 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  $\xrightarrow{\mathbf{e}}$  steps are forced to be followed by other  $\xrightarrow{\mathbf{e}}$  steps). We will not do this in this dissertation since it is rather straightforward.

**5.1.11 Proposition.** For any CRS  $R$ ,  $R\mathbf{x}$  is explicit.

**5.1.12 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.

**5.1.13 Proposition.** For a confluent CRS  $\rightarrow$ ,  $\xrightarrow{\mathbf{e}}$  is confluent and strongly normalising.

*Proof sketch.* Completeness of  $\xrightarrow{\mathbf{e}}$  is shown by first establishing  $\xrightarrow{\mathbf{e}}\text{SN}$  by defining a map dominating the longest reduction length similarly as for  $\lambda\mathbf{x}\text{gc}$  in Bloo and Rose (1995);  $\xrightarrow{\mathbf{e}}\text{LC}$  follows from a simple investigation of the critical pairs between the ( $\mathbf{x}\text{gc}\text{-n}$ ) rules and the other rules (this amounts to understanding that ‘garbage collection’ can be postponed without risk).  $\square$

<sup>2</sup>Generated automatically from ( $\beta$ ) by the program of 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  $\xrightarrow{e}\text{-nf}$  of  $t$ , and  $\xrightarrow{e}\gg$  the restriction of  $\xrightarrow{e}$  to reductions to normal forms).

**5.1.14 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  $\lambda$ -calculus in disguise; we prove by induction on the number of symbols in the sequence  $t, t_1, \dots, t_n$  that  $\downarrow_e(\Sigma([x_1, \dots, x_n]t, t_1, \dots, t_n)) \equiv \downarrow_e(t)[x_1 := \downarrow_e(t_1)] \cdots [x_n := \downarrow_e(t_n)]$ . We distinguish some cases according to the structure of  $t$ .

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

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

**Case  $t = F^m(\vec{s})$ .** Then  $\Sigma([\vec{x}]t, \vec{t}) \xrightarrow{e}\gg F^m(\Sigma([\vec{x}]s_1, \vec{t}), \dots, \Sigma([\vec{x}]s_m, \vec{t}))$ , hence

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

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

**5.1.15 Lemmas (projection & injection).**

A. For terms  $s, t$ ,  $\begin{array}{ccc} s & \xrightarrow{i} & t \\ \downarrow_e & & \downarrow_e \\ \downarrow_e(s) & \xrightarrow{\gg} & \downarrow_e(t) \end{array}$  . B. For pure terms  $s, t$ ,  $\begin{array}{ccc} s & \xrightarrow{\quad} & t \\ \downarrow_e & \searrow & \downarrow_e \\ i & \searrow & s' \xrightarrow{e} t \end{array}$  .

*Proof.* Lemma 5.1.15.A by induction over the structure of Rx-metaterms, using Lemma 5.1.14; Lemma 5.1.15.B is a simple consequence of the fact that the R-rules and Rx-introduction rules have the same redexes and the  $\xrightarrow{e}$ -rules construct no new redexes. The details are as follows:

- A. Induction over the structure of  $s$ . If the redex contracted in  $s_{\vec{i}}$  is at the root of  $s$ , then use Lemma 5.1.14. 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 5.1.14,  $\downarrow_e(s) \equiv \downarrow_e(s')[\vec{x} := \downarrow_e(\vec{s})]$  and by the induction hypothesis  $\downarrow_e(s_i) \twoheadrightarrow \downarrow_e(t_i)$  if  $t \equiv \Sigma([\vec{x}]t', \vec{t})$ .
- B. If  $s \twoheadrightarrow t$  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  $s_{\vec{i}} \twoheadrightarrow s'$ . Now by Lemma 5.1.14 and the definition of substitution in Definition 2.6.8 it is clear that  $\downarrow_e(s') \equiv t$ , hence  $s' \xrightarrow{e} t$ .  $\square$

**5.1.16 Theorem.** *For R-terms  $s, t$ ,  $s \xrightarrow{ie} t$  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 5.1.15:

**Case  $\Leftarrow$ :** Assume  $s \twoheadrightarrow t$ ; the case then follows by induction on the length of the  $\twoheadrightarrow$ -reduction, using Lemma 5.1.15.B in each step.

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

$$\begin{array}{ccccccc}
 s & \xrightarrow{ie} & s_1 & \xrightarrow{ie} & \cdots & \xrightarrow{ie} & s_{n-1} & \xrightarrow{ie} & t \\
 \parallel & & \downarrow e & & & & \downarrow e & & \parallel \\
 s & \twoheadrightarrow & \downarrow_e(s_1) & \twoheadrightarrow & \cdots & \twoheadrightarrow & \downarrow_e(s_{n-1}) & \twoheadrightarrow & t
 \end{array}$$

Each  $\xrightarrow{ie}$ -step is either  $\xrightarrow{i}$  or  $\xrightarrow{e}$  for which we need Lemma 5.1.15.A and confluence of  $\xrightarrow{e}$ , respectively (in the latter case  $\twoheadrightarrow$  is void).  $\square$

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

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

*Proof.* If  $t \xrightarrow{ie} s_1$  and  $t \xrightarrow{ie} s_2$  then by Lemma 5.1.15.A 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 5.1.16.  $\square$

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):

**5.1.18 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.

**5.1.19 Theorem.**  $Rx$  PSN of R.

*Proof.* See Bloo and Rose (1996). □

## 5.2 Explicit Addresses

In this section we show how the explicit sharing idea developed for the  $\lambda$ -calculus in section 3.5 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  $\lambda$ -calculus.

In this section we generalise the address notion of section 3.4 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  $\lambda$ -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.

**5.2.1 Definition (simple sharing CRS).** The set of *simply addressed CRS-preterms* over some alphabet is the set of preterms (as in Definition 2.6.1.B) extended with *addressing* forms (as in Definition 3.4.1.A), *i.e.*,

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

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 2.6.1. Matching and rewriting also proceeds quite as in Definition 2.6.12 with addresses seen as new function

symbols, except that the relations we define will always 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 3.4 up to Proposition 3.4.6 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 3.4.

### 5.2.2 Example (acyclic $\lambda$ -graph reduction). The CRSa

$$\begin{aligned} (\lambda x.Y(x))X &\rightarrow Y((X)^{\text{fresh}}) && (\beta a) \\ (\lambda x.Y(x))^Z &\rightarrow (\lambda x.Y(x)) && (\text{copy}) \end{aligned}$$

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

$$\begin{aligned} & \boxed{(\lambda k.(\lambda abc.ac(bc))kk)((\lambda a.a)(\lambda a.a))(\lambda ab.a)} \\ & \quad \downarrow (\beta a) \\ & \boxed{((\lambda abc.ac(bc))(((\lambda a.a)(\lambda a.a))(\lambda ab.a))^1)} \boxed{(((\lambda a.a)(\lambda a.a))(\lambda ab.a))^1} \\ & \quad \downarrow (\beta a) \\ & \boxed{(\lambda ab.((((\lambda a.a)(\lambda a.a))(\lambda ab.a))^2)b(ab))} \boxed{(((\lambda a.a)(\lambda a.a))(\lambda ab.a))^1} \\ & \quad \downarrow (\beta a) \\ & \lambda a.(\boxed{((\lambda b.b)(\lambda b.b))}(\lambda bc.b)^1)^2 \boxed{a} \boxed{(((\lambda a.a)(\lambda a.a))(\lambda ab.a))^1}^3 \boxed{a} \\ & \quad \downarrow (\beta a) \\ & \lambda a.(\boxed{((\lambda b.b)^4)}(\lambda bc.b)^1)^2 \boxed{a} \boxed{(((\lambda b.b)^4)}(\lambda bc.b)^1)^3 \boxed{a} \\ & \quad \downarrow (\text{copy}) \\ & \lambda a.(\boxed{((\lambda b.b)(\lambda bc.b))}^1)^2 \boxed{a} \boxed{(((\lambda b.b)(\lambda bc.b))}^1)^3 \boxed{a} \\ & \quad \downarrow (\beta a) \\ & \lambda a.(\boxed{((\lambda bc.b)^6)}^1)^2 \boxed{a} \boxed{(((\lambda bc.b)^6)}^1)^3 \boxed{a} \\ & \quad \downarrow (\text{copy}) \\ & \lambda a.(\boxed{((\lambda bc.b)^1)}^1)^2 \boxed{a} \boxed{(((\lambda bc.b)^1)}^1)^3 \boxed{a} \\ & \quad \downarrow (\text{copy}) \end{aligned}$$

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

As for the  $\lambda$ -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.

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

For such a system everything works out nicely, just as for the  $\lambda$ -calculus in section 3.5: since all rewrites are local no undesired ‘side effects’ happen in the term. We will demonstrate this, but first we will strengthen the notion of sharing.

**5.2.4 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  $\lambda$ -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.



**5.2.5 Definition (cyclic sharing CRS).** The set of *cyclic addressed CRS-preterms* over some alphabet is the set of simple sharing preterms (as in Definition 5.2.1) extended with *cyclic addressing* forms, *i.e.*,

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

Rewriting happens as for CRSa except we add another metanotation that is allowed in rule RHSs, namely  $t_1 \parallel t_2^a$  which means  $t_1[t_2^a]$  in the notation of Definition 3.4.2.B 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.

**5.2.6 Definitions (cyclic sharing terms).** Assume a set of addressed T-terms as above.

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

**5.2.7 Example (cyclic  $\lambda$ -graph reduction).** The CRSa

$$\begin{aligned} (\lambda x.Y(x))X &\rightarrow Y((X)^{\text{fresh}}) && (\beta a) \\ (\lambda x.Y(x))^Z &\rightarrow (\lambda x.Y(x)) && (\text{copy}) \end{aligned}$$

defines  $\lambda$ -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  $\mu$ -rule  $\mu x.Z(x) \rightarrow Z(\mu x.Z(x))$  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 5.3.

**5.2.8 Comparison (cyclic  $\lambda$ -graph reduction).** The  $\lambda$ -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 dissertation 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)  $\lambda$ -graphs is that *cycles remain observable*: it is possible to write rules that manipulate  $\bullet$ 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.

## 5.3 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 2.3.11 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, van Eekelen, Glauert, Kennaway, Plasmeijer and Sleep (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  $\lambda$ -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 demonstrate 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  $\lambda$ -calculus).

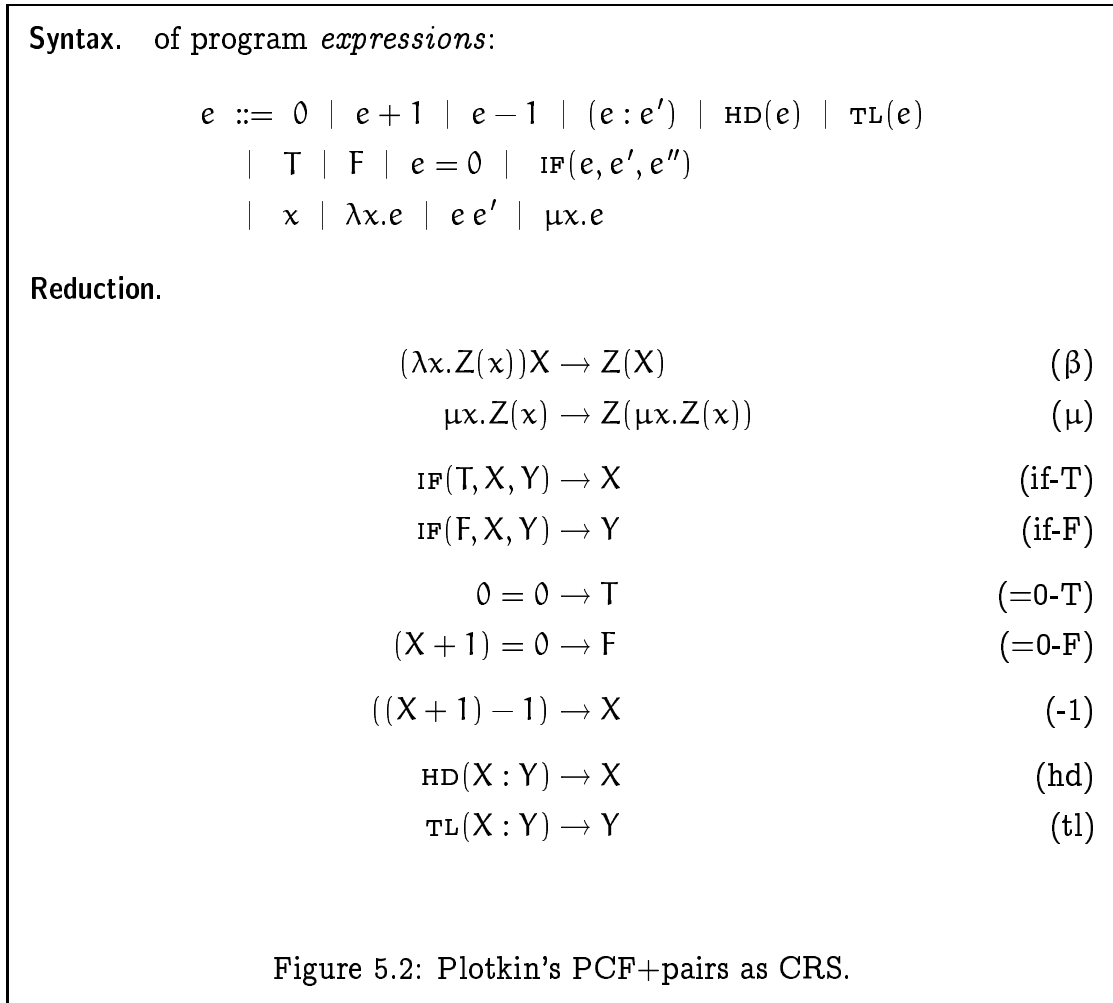
In a nutshell: what is ‘wrong’ with, say the differential rewrite system of Figure 2.2, if we want to implement it? Three things pop up as ‘expensive’, namely

**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  $\lambda$ -calculus in chapter 3.

The first aspect was already defined in section 2.6, 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  $\text{HD}$  and  $\text{TL}$  selectors, and show how sharing and explicification works.

The PCF+pairs programs and evaluation reduction system as a CRS are given in Figure 5.2. The following small program computes the parity of a number, or rather: the pair of booleans (*is-even* : *is-odd*).

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

Here is the start and finish of a reduction sequence computing<sup>3</sup> the even and

<sup>3</sup>Computed by the program of chapter 6 acting on Figure 5.2.

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).

$$\begin{array}{c}
\boxed{(\mu f.\lambda n.(\text{IF}(n = 0, \text{T}, \text{TL}(f(n - 1)))) : \text{IF}(n = 0, \text{F}, \text{HD}(f(n - 1))))} \boxed{(((0 + 1) + 1) + 1)} \\
(\mu) \\
\downarrow \\
\boxed{(\lambda a.(\text{IF}(a = 0, \text{T}, \text{TL}((\mu a.\lambda b.(\text{IF}(b = 0, \text{T}, \text{TL}(a(b - 1)))) : \text{IF}(b = 0, \text{F}, \text{HD}(a(b - 1))))(a - 1)))) : \text{IF}(a = 0, \text{F}, \text{HD}((\mu a.\lambda b.(\text{IF}(b = 0, \text{T}, \text{TL}(a(b - 1)))) : \text{IF}(b = 0, \text{F}, \text{HD}(a(b - 1))))(a - 1))))((0 + 1) + 1) + 1)} \\
(\beta) \\
\downarrow \\
(\text{IF}(\boxed{(((0 + 1) + 1) + 1) = 0}, \text{T}, \text{TL}(\boxed{(\mu a.\lambda b.(\text{IF}(b = 0, \text{T}, \text{TL}(a(b - 1)))) : \text{IF}(b = 0, \text{F}, \text{HD}(a(b - 1))))})) \\
(=0-F) \\
\boxed{(\mu a.\lambda b.(\text{IF}(b = 0, \text{T}, \text{TL}(a(b - 1)))) : \text{IF}(b = 0, \text{F}, \text{HD}(a(b - 1))))} \\
(\mu) \\
\boxed{(((0 + 1) + 1) + 1) - 1} : \text{IF}(\boxed{(((0 + 1) + 1) + 1) = 0}, \text{F}, \text{HD}(\boxed{(\mu a.\lambda b.(\text{IF}(b = 0, \text{T}, \text{TL}(a(b - 1)))) : \text{IF}(b = 0, \text{F}, \text{HD}(a(b - 1))))})) \\
(-1) \quad (=0-F) \\
\boxed{(\mu a.\lambda b.(\text{IF}(b = 0, \text{T}, \text{TL}(a(b - 1)))) : \text{IF}(b = 0, \text{F}, \text{HD}(a(b - 1))))} \boxed{(((0 + 1) + 1) + 1) - 1} \\
(\mu) \quad (-1) \\
\Downarrow \\
\vdots \\
\Downarrow \\
(\text{IF}(\boxed{0 = 0}, \text{F}, \text{HD}(\boxed{(\mu a.\lambda b.(\text{IF}(b = 0, \text{T}, \text{TL}(a(b - 1)))) : \text{IF}(b = 0, \text{F}, \text{HD}(a(b - 1))))} (0 - 1)))) : \\
(=0-T) \quad (\mu) \\
\text{IF}(\boxed{0 = 0}, \text{T}, \text{TL}(\boxed{(\mu a.\lambda b.(\text{IF}(b = 0, \text{T}, \text{TL}(a(b - 1)))) : \text{IF}(b = 0, \text{F}, \text{HD}(a(b - 1))))} (0 - 1))) \\
(=0-T) \quad (\mu) \\
\Downarrow \\
\boxed{(\text{IF}(\text{T}, \text{F}, \text{HD}((\lambda a.(\text{IF}(a = 0, \text{T}, \text{TL}((\mu a.\lambda b.(\text{IF}(b = 0, \text{T}, \text{TL}(a(b - 1)))) : \text{IF}(b = 0, \text{F}, \text{HD}(a(b - 1))))(a - 1)))) : \text{IF}(b = 0, \text{F}, \text{HD}(a(b - 1))))(a - 1)))) : \text{IF}(a = 0, \text{F}, \text{HD}((\mu a.\lambda b.(\text{IF}(b = 0, \text{T}, \text{TL}(a(b - 1)))) : \text{IF}(b = 0, \text{F}, \text{HD}(a(b - 1))))(a - 1))))(0 - 1)))} \\
(\text{if-T}) \\
\boxed{(\text{IF}(\text{T}, \text{T}, \text{TL}((\lambda a.(\text{IF}(a = 0, \text{T}, \text{TL}((\mu a.\lambda b.(\text{IF}(b = 0, \text{T}, \text{TL}(a(b - 1)))) : \text{IF}(b = 0, \text{F}, \text{HD}(a(b - 1))))(a - 1)))) : \text{IF}(b = 0, \text{F}, \text{HD}(a(b - 1))))(a - 1)))) : \text{IF}(a = 0, \text{F}, \text{HD}((\mu a.\lambda b.(\text{IF}(b = 0, \text{T}, \text{TL}(a(b - 1)))) : \text{IF}(b = 0, \text{F}, \text{HD}(a(b - 1))))(a - 1))))(0 - 1)))} \\
(\text{if-T}) \\
\Downarrow \\
\text{F} : \text{T}
\end{array}$$

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

$$\begin{aligned}
 e ::= & 0 \mid e + 1 \mid e - 1 \mid (e : e') \mid \text{HD}(e) \mid \text{TL}(e) \\
 & \mid \text{T} \mid \text{F} \mid e = 0 \mid \text{IF}(e, e', e'') \\
 & \mid x \mid \lambda x.e \mid e e' \mid \mu x.e \\
 & \mid e^a
 \end{aligned}$$

where  $a$  ranges over addresses.

**Reduction.**

$$\begin{aligned}
 (\lambda x.Z(x))X &\rightarrow Z((X)^a) && (\beta) \\
 (Z)^V X &\rightarrow (Z \parallel (Z)^V)X && (\beta\text{-copy}) \\
 \mu x.Z(x) &\rightarrow (Z((\blacksquare)^a))^a && (\mu) \\
 \text{IF}(\text{F}, X, Y) &\rightarrow Y && (\text{if-F}) \\
 \text{IF}(\text{T}, X, Y) &\rightarrow X && (\text{if-T}) \\
 \text{IF}((Z)^V, X, Y) &\rightarrow \text{IF}(Z \parallel (Z)^V, X, Y) && (\text{if-copy}) \\
 0 = 0 &\rightarrow \text{T} && (=0-T) \\
 (X + 1) = 0 &\rightarrow \text{F} && (=0-F) \\
 (Z)^V = 0 &\rightarrow (Z \parallel (Z)^V) = 0 && (=0-copy) \\
 \text{HD}(X : Y) &\rightarrow X && (\text{hd}) \\
 \text{HD}((Z)^V) &\rightarrow \text{HD}(Z \parallel (Z)^V) && (\text{hd-copy}) \\
 \text{TL}(X : Y) &\rightarrow Y && (\text{tl}) \\
 \text{TL}((Z)^V) &\rightarrow \text{TL}(Z \parallel (Z)^V) && (\text{tl-copy}) \\
 ((X + 1) - 1) &\rightarrow X && (-1) \\
 (Z)^V - 1 &\rightarrow (Z \parallel (Z)^V) - 1 && (-1-copy)
 \end{aligned}$$

Figure 5.3: PCF+pairs with cyclic sharing as CRSar.

$(\lambda x. Z(x))X \rightarrow \Sigma([x]Z(x), (X)^a)$	( $\beta$ -x)
$\mu x. Z(x) \rightarrow (\Sigma([x]Z(x), (\blacksquare)^a))^a$	( $\mu$ -x)
$(Z)^X \rightarrow (Z \square (Z)^X)$	(copy)
$\text{IF}(F, X, Y) \rightarrow Y$	(if-F)
$\text{IF}(T, X, Y) \rightarrow X$	(if-T)
$0 = 0 \rightarrow T$	(=0-T)
$(X + 1) = 0 \rightarrow F$	(=0-F)
$\text{HD}((X : Y)) \rightarrow X$	(hd)
$\text{TL}((X : Y)) \rightarrow Y$	(tl)
$(X + 1) - 1 \rightarrow X$	(-1)
$\Sigma([x](Z_1(x))(Z_2(x)), X) \rightarrow \Sigma([x]Z_1(x), X)\Sigma([x]Z_2(x), X)$	(x-@-1)
$\Sigma([xy]Z(x, y), X) \rightarrow [y]\Sigma([x]Z(x, y), X)$	(xma-1)
$\Sigma([x]\lambda(Z(x)), X) \rightarrow \lambda(\Sigma([x]Z(x), X))$	(x- $\lambda$ -1)
$\Sigma([x]\mu(Z(x)), X) \rightarrow \mu(\Sigma([x]Z(x), X))$	(x- $\mu$ -1)
$\Sigma([x]\text{IF}(Z_1(x), Z_2(x), Z_3(x)), X) \rightarrow \text{IF}(\Sigma([x]Z_1(x), X), \Sigma([x]Z_2(x), X), \Sigma([x]Z_3(x), X))$	(x-IF-1)
$\Sigma([x]F, X) \rightarrow F$	(x-F-1)
$\Sigma([x]T, X) \rightarrow T$	(x-T-1)
$\Sigma([x](Z_1(x)) = (Z_2(x)), X) \rightarrow (\Sigma([x]Z_1(x), X)) = (\Sigma([x]Z_2(x), X))$	(x==1)
$\Sigma([x]0, X) \rightarrow 0$	(x-0-1)
$\Sigma([x](Z_1(x)) + (Z_2(x)), X) \rightarrow (\Sigma([x]Z_1(x), X)) + (\Sigma([x]Z_2(x), X))$	(x+-1)
$\Sigma([x]1, X) \rightarrow 1$	(x-1-1)
$\Sigma([x]\text{HD}((Z(x))), X) \rightarrow \text{HD}((\Sigma([x]Z(x), X)))$	(x-HD-1)
$\Sigma([x](Z_1(x)) : Z_2(x), X) \rightarrow (\Sigma([x]Z_1(x), X)) : \Sigma([x]Z_2(x), X)$	(x:-1)
$\Sigma([x]\text{TL}((Z(x))), X) \rightarrow \text{TL}((\Sigma([x]Z(x), X)))$	(x-TL-1)
$\Sigma([x](Z_1(x)) - (Z_2(x)), X) \rightarrow (\Sigma([x]Z_1(x), X)) - (\Sigma([x]Z_2(x), X))$	(x--1)
$\Sigma([x]x, X) \rightarrow X$	(xv-1-1)
$\Sigma([x]Z, X) \rightarrow Z$	(xgc-1-1)

Figure 5.4: PCF+pairs as CRSar after explicification of substitutions.

This is the desired result, of course, since 3 is odd, not even. The full sequence of 16 steps is shown in Example A.3.1 on page 217; reducing only the leftmost redex the same reduction takes 48 steps.

Finally we show the language as a CRSar before and after explicit substitution in Figure 5.3 and Figure 5.4, respectively; executions using these systems are shown in Example A.3.2 and (outlined) in Code A.3.3, respectively.

The first, the CRSa version, just does the same kind of updating as we did for the  $\lambda$ -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).

## 5.4 Summary

This section 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).

variant	bounds	section
explicit substitution	rewrite time	5.1
saturation & free variables	matching time	5.1
sharing & recursion	copying space	5.2

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:

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



- B. Use this information to perform *program transformations* that are ‘safe’ in the sense that they are *correct* and guaranteed to *terminate*.
- C. Discard the analysis information!
- D. 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! The only caveat is that the calculus must have useable operational properties with respect to both aspects in the sense discussed above.

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.

We are grateful to Roel Bloo for collaboration on ESCRS, including contributing the recent Theorem 5.1.19.



# 6

## Implementation of Combinatory Reduction Systems

In this chapter we present the complete Haskell (Hudak, Peyton Jones, Wadler et al. 1992) implementation of a program that does CRS reduction as defined in section 2.6 extended with some of the transformations described in chapter 5.

CRS reduction is a very powerful paradigm, and little emphasis has been put on implementation of such systems – the only implementation we know of is the compiler of Kahrs (1993) which only treats a subset of CRS reduction corresponding to what functional programs can do, and compiles it into an abstract machine (!) with powerful instructions specifically designed for CRS matching. Our implementation of matching is naïve in comparison, however, we include more ‘frills’ in the sense of allowing analysis of and experimentation with strategies *etc*, and the implementation of our extensions is new; in particular transformations of rules require a ‘direct’ representation such as we use.

We first present in section 6.1 the notation of literate programming and some generic functions we will make frequent use of. We then proceed in section 6.2 with the declaration of the CRS datatype and several associated meth-

ods, notably the output functions, in section 6.3 the input parser follows. Section 6.4 is the central CRS section with the implementation of reduction; it is followed by section 6.5 where we have specified some useful reduction strategies and section 6.6 where we have coded the tests on CRSs that we use. Section 6.7 implements *explicitification* as handled in section 5.1, and section 6.8 adds an *inference rule* interpretation (not described elsewhere as it merely pushes tokens around to emulate a strategy). Finally, section 6.9 contains the code needed to bind the rest together.

## 6.1 Idioms

In this section we present a few idiomatic definitions that make the remainder of the system smoother.<sup>1</sup>

**6.1.1 Notation (literate Haskell).** Lines indented with a small number  $1\ 2\ \dots$  in this dissertation are *literate Haskell* source lines (the traditional character used for such indentation is the obtrusive `>`, and in fact this character *is* used in the files but we typeset it with line numbers for reference). Furthermore we use the special symbols  $\rightarrow$ ,  $\leftarrow$ ,  $\equiv$ ,  $\vee$ ,  $\wedge$ ,  $\neq$ ,  $\geq$ ,  $\leq$ , and  $\lambda$ , as readable alternatives to the standard Haskell symbols `->`, `<-`, `==`, `||`, `&&`, `/=`, `>=`, `<=`, and `\`, and we use `□` for spaces inside Haskell strings.

**6.1.2 Code (tracing).** We only use the `trace` primitive in a non-obtrusive way, printing ‘comment-marked’ things out that cannot show up in the result. Two versions are included: one for a string message and one that invokes ‘`show`’ on the argument (the check of the length is only there to make the definition strict regardless of whether the primitive is loaded).

```

1 trc :: String → a → a
2 trc s x | length s ≥ 0 = trace s x

3 tr :: String → a → a
4 tr s x | length s ≥ 0 = trace ("%%" ++ s ++ "\n") x

5 tm :: Text a => String → a → a
6 tm s x = tr (s ++ show x) x
```

---

<sup>1</sup>The section is the result of typesetting the literate Haskell script “idioms.lhs”.

**6.1.3 Code (checking).** Most predicates are implemented as what we will call a “check” meaning that it returns a string with the *failure message* (so success is indicated by “”).

```
1 type Check = String
```

Disjunction of checks is handled by the `either` (conjunction is just `++`).

```
2 either :: Check → Check → Check
3 either c1 c2 = case (c1,c2) of ("","") → ""
4                               (_, "") → ""
5                               ("",_) → ""
6                               _       → c1 ++ c2
```

The `check` function turns a check into a boolean predicate (the opposite is easiest accomplished with an `if` statement since an explanation for the failure is needed).

```
7 check :: Check → Bool
8 check "" = True
9 check _  = False
```

The `assert` function aborts with an error if a check failed.

```
10 assert :: Check → a → a
11 assert "" v = v
12 assert es _ = error es
```

**6.1.4 Code (sets).** We will implement ‘sets’ as lists, in fact set generating predicates over terms will produce lists with elements ordered as the occurrences from left to right (by using plain list concatenation `++` to implement  $\cup$ ) – essentially these are ‘occurrence-ordered multisets’. The only operation that is not obvious is set difference.

```
1 but :: Eq a => [a] → [a] → [a]
2 but xs []      = xs
3 but xs (y:ys) = but (filter (≠ y) xs) ys
```

This implementation has the advantage that one can check for ‘duplicates’.

```
4 duplicates []      = []
5 duplicates (x:xs) | x 'elem' xs = x : rest
6                   | otherwise  = rest
7                   where rest = duplicates (xs 'but' [x])
```

**6.1.5 Code (translation maps).** These are (usually small) ‘finite maps’ encoded using lookup such that there is quick access to the range and domain.

```
1 data Trans a = Trans [a] [a]
2 trans :: Eq a => Trans a -> a -> a
```

We allow the ‘range list’ to be longer such that it can be generated lazily.

```
3 trans (Trans [] _ ) x = x
4 trans (Trans(x1:x1s)(x2:x2s)) x | x1 ≡ x = x2
5 | otherwise = trans (Trans x1s x2s) x
6 trans1 (Trans x1s _ ) = x1s
7 trans2 (Trans _ x2s) = x2s
```

The domain and range can be queried with these predicates.

```
8 hastrans tr x = x `elem` (trans1 tr)
9 istrans tr x = x `elem` (trans2 tr)
```

And the inverse can be constructed.

```
10 inversetrans (Trans x1s x2s) = Trans x2s x1s
```

Finally some constructor functions.<sup>2</sup>

```
11 notrans = Trans [] []
12 mktrans x1s x2s = Trans x1s x2s
13 xtrans x1 x2 (Trans x1s x2s) = (Trans (x1:x1s) (x2:x2s))
```

And we permit printing of translations.

```
14 instance (Text a) => Text (Trans a) where
15   showsPrec d (Trans [] _) rest = "\\{\\}"
16   showsPrec d (Trans x1s x2s) rest = "\\{"
17     ++foldr1 (λs1 s2 → s1++";␣"++s2)
18       (zipWith (λx1 x2 → uq(show x1)+"->"++uq(show x2)) x1s x2s)
19     ++"\\}"++rest
20   where { uq ('':s) = init s ; uq s = s }
```

**6.1.6 Code (sorting).** We include a generic quicksort function (that also removes duplicates).

```
1 sort :: Ord a => [a] -> [a]
2 sort [] = []
3 sort (x:xs) = sort [ u | u ← xs, u < x ]
```

---

<sup>2</sup>Trans should probably be a class.

```

4         ++ [x]
5         ++ sort [ u | u ← xs, u > x ]

```

**6.1.7 Code (i/o errors).** Are handled too silent by the standard exit command ... this is more noisy.

```

1 complain :: FailCont
2 complain (ReadError s) = error ("ReadError_on_'" ++ s ++ "'")
3 complain (WriteError s) = error ("WriteError_on_'" ++ s ++ "'")
4 complain (SearchError s) = error ("SearchError_on_'" ++ s ++ "'")
5 complain (FormatError s) = error ("FormatError_on_'" ++ s ++ "'")
6 complain (OtherError s) = error ("OtherError_on_'" ++ s ++ "'")

```

## 6.2 Datatypes

In this section we present the *CRS datatype*,<sup>3</sup> *i.e.*, the syntactic issues of section 2.6: we follow closely the conventions from Definition 2.6.1 through Notation 2.6.5) as as possible, followed by the printing functions (the input functions are presented separately in section 6.3).

**6.2.1 Code (function symbols).** Name and arity of Definition 2.6.1.A plus a flag used for the following purposes:

- For function symbols it means that the symbol should be printed using *infix notation*.
- For zero-ary metavariables it means that the metavariable is subject to the *free variable matching constraint*.
- For other metavariables it means it is used to indicate that the variable is *saturated* (this makes matching faster).

```

1 type Sym = (String, Int, Bool)

```

**6.2.2 Code (variables).** Variables are merely strings.

```

1 type Var = String

```

---

<sup>3</sup>The section is the result of typesetting the literate Haskell script “crstype.lhs”.

We often need ‘fresh’ variables which are realised by an infinite list of variables that are ‘fresh’ in some context. Only single letters are used and we prefer them without a suffix (and written in  $\text{\TeX}$  notation, all as accepted by the CRS parser).

```

2 fresh :: [Var] → [Var]
3 fresh vs = filter ('notElem' vs)
4           ([ [v] | v ← ['a'..'z'] ]
5            ++ [ [v] ++ "_" ++ shownumb n | v ← ['a'..'z'], n ← [1..9] ]
6            ++ [ [v] ++ "_" ++ shownumb n | n ← [10..], v ← ['a'..'z'] ])

```

The following is extracted because it will be used later.

```

7 shownumb n = if n < 10 then show n else "{" ++ show n ++ "}"

```

**6.2.3 Code (preterms).** The `Mterm` datatype is used to represent metaterms; it has a constructor for each form of preterm from Definition 2.6.1.B; metavariables are treated as function symbols because they have an arity.

```

1 type Mv      = Sym
2 data Mterm = Mvar Var
3             | Mabs Var Mterm
4             | Mcon Sym [Mterm]
5             | Mapp Mv  [Mterm]

```

**6.2.4 Code (metavariables).** The *set of metavariables*  $mv(t)$  of Definition 2.6.1.C.

```

1 mv :: Mterm → [Sym]
2 mv (Mvar v)      = []
3 mv (Mabs v t)    = mv t
4 mv (Mcon f ts)   = concat [ mv t | t ← ts ]
5 mv (Mapp m ts)   = m : concat [ mv t | t ← ts ]

```

**6.2.5 Code (free variables).** The *set of free variables*  $fv(t)$  of Definition 2.6.1.D.

```

1 fv :: Mterm → [Var]
2 fv (Mvar v)      = [v]
3 fv (Mabs v t)    = filter (≠ v) (fv t)
4 fv (Mcon f ts)   = concat [ fv t | t ← ts ]
5 fv (Mapp m ts)   = concat [ fv t | t ← ts ]

```

The `checkMterm` predicate verifies that a `meta(pre)term` is *closed*.



```

6 checkMterm :: Mterm → Check
7 checkMterm t = case fv t of
8     [] → ""
9     vs → "\Metaterm_Error:unbound_vars"
10        ++concat[" | v ← vs] ++".

```

**6.2.6 Code (metaterm equality).** Equality of metaterms in Definition 2.6.1.E is modulo renaming –  $t[x := y]$  is realised by  $\text{rename}(\text{mktrans}[x][y])(t)$  but in general several renamings can happen simultaneously (renaming uses translation maps).

```

1 rename :: Trans Var → Mterm → Mterm
2 rename tr (Mvar v)    = Mvar (trans tr v)
3 rename tr (Mabs v t) = Mabs v (rename (xtrans v v tr) t)
4 rename tr (Mcon f ts) = Mcon f (map (rename tr) ts)
5 rename tr (Mapp m ts) = Mapp m (map (rename tr) ts)

```

The actual equality is defined as follows using translation maps directly (for efficiency) and such that  $\equiv$  on (meta)terms means  $\equiv$ , *i.e.*,  $\alpha$ -equivalence;

```

6 instance Eq Mterm where (≡) = teq notrans where
7   teq vts (Mvar v)    (Mvar v')    = trans vts v ≡ v'
8   teq vts (Mabs v t) (Mabs v' t') = teq (xtrans v v' vts) t t'
9   teq vts (Mcon f ts) (Mcon f' ts') = f ≡ f' ∧ tseq vts ts ts'
10  teq vts (Mapp m ts) (Mapp m' ts') = m ≡ m' ∧ tseq vts ts ts'
11  teq _ _ _ _ _ _ _ _ _ _ _ _ _ _ = False
12  tseq vts [] [] _ _ _ _ _ _ _ _ = True
13  tseq vts (t:ts) (t':ts') = tseq vts ts ts' ∧ teq vts t t'
14  tseq _ _ _ _ _ _ _ _ _ _ _ _ _ = False

```

**6.2.7 Code (rewrite rules).** Rules are the LHS and RHS metaterms as defined in Definition 2.6.1.G and a name for printing.

```

1 type Rule = (Mterm,Mterm,String)

```

We check the LHS and RHS separately.

```

2 checkRule :: Rule → Check
3 checkRule r@(_,_,nm) =
4   either ("\nRule_++nm+_LHS_errors:_") (checkLHS r)
5   ++ either ("\nRule_++nm+_RHS_errors:_") (checkRHS r)

```

The LHS checks are as follows.

```

6  checkLHS :: Rule → Check
7  checkLHS (lhs@(Mcon _ _),_,_) =
8    checkMterm lhs ++ checkpattern lhs where
9      checkpattern :: Mterm → Check
10     checkpattern (Mvar v)      = ""
11     checkpattern (Mabs v t)    = checkpattern t
12     checkpattern (Mcon f ts)   = concat [ checkpattern t | t ← ts ]
13     checkpattern (Mapp m ts)   = checkdistinctvars [] ts
14     checkdistinctvars :: [Var] → [Mterm] → Check
15     checkdistinctvars _ []     = ""
16     checkdistinctvars vs (Mvar v:ts) =
17       (if v 'elem' vs then "\n␣Repeated␣var:␣" ++ v ++ "." else "")
18       ++ checkdistinctvars vs ts
19     checkdistinctvars vs (t      :ts) =
20       "\n␣Non-var␣argument␣" ++ showMterm t ++ "␣in␣pattern."
21       ++ checkdistinctvars vs ts
22 checkLHS (lhs,_,_) =
23   "\n␣Pattern␣" ++ showMterm lhs ++ "␣is␣not␣a␣construction."

```

The RHS checks are easier.

```

24 checkRHS :: Rule → Check
25 checkRHS (lhs,rhs,_) =
26   checkMterm rhs
27   ++
28   (case (mv rhs) 'but' (mv lhs) of
29     [] → ""
30     ms → "\n␣Metavar(s)␣" ++ concat [showSym' m ++ "␣" | m ← ms]
31     ++ "␣only␣in␣RHS.")

```

**6.2.8 Code (CRS).** Finally, entire CRS systems are encoded directly as lists of rules as prescribed by Definition 2.6.1.

```
1  type CRS = [Rule]
```

This means that checking a CRS is merely checking that the (obligatory) rules are correct.

```

2  checkCRS :: CRS → Check
3  checkCRS crs = either "Errors␣in␣CRS:␣" messages where
4    messages = concat [ checkRule r | r ← crs ]

```

```

5         ++concat (nub [ "\n␣Duplicated␣rule␣name␣" ++nm++ ". "
6                   | nm ← rulenames, nm 'elem' (rulenames \\ [nm]) ])
7  rulenames = [ (λ(␣,␣, nm) → nm) r | r ← crs ]

```

The remainder of this section is concerned with *printing CRS* symbols, metaterms and rules. These are generally quite complicated since an effort has been put into producing output that looks pleasing when processed by the  $\text{T}_{\text{E}}\text{X}$  typesetter (Knuth 1984) using the  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$  format (Lamport 1994) and the special math abbreviation package `qsymbols` (Rose 1994).

**6.2.9 Code (printing function symbols).** Printable representation of symbols (in two versions: short and verbose; in both cases in  $\text{T}_{\text{E}}\text{X}$  form).

```

1  showSym, showSym' :: Sym → String
2  showSym = showSymP True
3  showSym' = showSymP False
4  showSymP p (s,a,␣) =
5    (if s ≡ "^" then "␣^" else s)
6    ++if p ∨ a < 0
7      then if head s ≡ '\ ' ^ all isletter(tail s) then "␣" else ""
8      else "^" ++if a < 10 then show a
9              else "{" ++show a ++ "}"
10 isletter c = c 'elem' (['a'..'z'] ++ ['A'..'Z'])

```

**6.2.10 Code (printing metaterms).** This rather long-winded definition prints a metaterm, using all sorts of shorthands understood by the parser as introduced in Notation 2.6.5.

```

1  showMterm :: Mterm → String
2  showMterm = showMtermP True
3  showMtermP p t = showMterm' t where
4    showMterm' (Mvar v)      = v
5    showMterm' (Mabs v t)    = showMabs v t
6    showMterm' (Mcon fs ts) = showMcon fs ts p
7    showMterm' (Mapp mv ts) = showSymP p mv ++ showMtermlist ts
8    showMabs vs (Mabs v t) = showMabs (vs ++ v) t
9    showMabs vs t          = "[" ++ vs ++ "]" ++ showMterm' t
10   showMcon fs@(s,1,False) [Mabs v t] True = showMcon' fs v t
11   showMcon fs@(s@('':_),2,True) [t1,t2] True =

```

```

12           "␣{{␣"++showSimple' t1++"␣"++s++"␣"++showSimple' t2++"␣}}␣"
13 showMcon fs@(":",2,True) [t1,t2]   True = showSimple t1++":"++showMterm' t2
14 showMcon fs@("@",2,_) [s,t]       True = showApp s t
15 showMcon fs@("^",2,True) [s,t]     _ = "("++showMterm' s++")^"++showSimple t
16 showMcon fs@(s,2,True) [t1,t2]    True = showSimple t1++s++showSimple t2
17 showMcon fs@(s@('\'\'X':_),_,False)ts True = s++"␣{{␣"++showMtermlist' ts++"␣}}␣"
18 showMcon fs@("{,}",0,False) []     True = "()"
19 showMcon fs@("{,}",n+2,False) ts   True = showMtermlist ts
20 showMcon fs@("<>",0,False) []      True = "<>"
21 showMcon fs@("<>",n+1,False) ts    True = "<"++showMtermlist' ts++">"
22 showMcon fs ts _ = showSymP p fs++showMtermlist ts

23 showMcon' fs@(s,1,False) vs t'@(Mcon (s',1,False) [Mabs v t])
24     | s ≡ s' = showMcon' fs (vs+v) t
25     | otherwise = showSymP p fs+vs++"."++showMterm' t'
26 showMcon' fs vs t' = showSymP p fs+vs++"."++showMterm' t'

27 showMcon'' fs@("@",2,_) ts = "("++showMcon fs ts p++")"
28 showMcon'' fs@(s,1,False) [Mabs v t] = "("++showMcon' fs v t++")"
29 showMcon'' fs@(_,a,False) ts | a ≡ 0 = "("++showMcon fs ts p++")"
30                               | otherwise = showMcon fs ts p
31 showMcon'' fs ts = "("++showMcon fs ts p++")"

32 showSimple' (Mcon (s@('\'':_),2,True) [t1,t2]) =
33     "␣{{␣"++showSimple t1++s++showSimple t2++"␣}}␣"
34 showSimple' t = showSimple t

35 showSimple (Mvar v) = v
36 showSimple (Mabs v t) = "("++showMabs v t++")"
37 showSimple (Mcon fs []) = showSymP p fs
38 showSimple (Mcon ("@",2,_) [s,t]) = showApp s t
39 showSimple (Mcon ("^",2,True) [s,t]) =
40     "("++showMterm' s++")^"++showSimple t
41 showSimple (Mcon ("{}",0,False) []) = "()"
42 showSimple (Mcon ("{}",n+2,False) ts) = showMtermlist ts
43 showSimple (Mcon ("<>",0,False) []) = "<>"
44 showSimple (Mcon ("<>",n,False) ts) = "<"++showMtermlist' ts++">"
45 showSimple (Mcon fs ts) = "("++showMcon fs ts p++")"
46 showSimple (Mapp mv []) = showSymP p mv
47 showSimple (Mapp mv ts) = "("++showSymP p mv++showMtermlist ts++")"

48 showApp s (Mvar v) = showB4var s+v
49 showApp s (Mcon fs []) = showSimple s++showSymP p fs
50 showApp s t@(Mcon fs ts) = showB4par s++showMcon'' fs ts
51 showApp s (Mapp mv []) = showSimple s++showSymP p mv
52 showApp s t = showB4par s++ "("++showMterm' t++")"
53 showB4par (Mvar v) = v

```

```

54 showB4par (Mcon ("@",2,_) [s,Mvar v]) = showB4var s++v
55 showB4par t@(Mcon fs@(_,a,False) ts) = showMcon' ' fs ts
56 showB4par t                          = "("++showMterm' t++")"
57 showB4var (Mvar v)                    = v
58 showB4var (Mapp mv [])                = showSymP p mv
59 showB4var (Mcon fs [])                = showSymP p fs
60 showB4var (Mcon ("@",2,_) [s,Mvar v]) = showB4var s++v
61 showB4var t                          = "("++showMterm' t++")"
62 showMtermlist [] = ""
63 showMtermlist ts = "("++showMtermlist' ts++")"
64 showMtermlist' [t] = showMterm' t
65 showMtermlist' (t:ts) = showMterm' t++","++showMtermlist' ts

```

**6.2.11 Code (printing CRS rule).** We exploit the metaterm printing functions to print a rule in a form readable by humans as well as processable by  $\text{\LaTeX}$  and rereadable by the parser.

```

1 showRule :: Rule → String
2 showRule = showRuleP True
3 showRuleP p (l,r,i) =
4   "␣"++showMtermP p l
5   ++ "␣&\"->\"␣"++showMtermP p r
6   ++ "␣\\tag{\"+i+\"}"

```

**6.2.12 Code (printing CRS).** Printing a CRS is merely printing the rules.

```

1 showCRS :: CRS → String
2 showCRS = showCRSP True
3 showCRSP p [] = "%CRS:;%END\n"
4 showCRSP p rs = "%CRS:\n"++showRules rs++"%END\n" where
5   showRules [r] = showRuleP p r++"\n"
6   showRules (r:rs) = showRuleP p r++"\\\\\n"++showRules rs

```

## 6.3 Input

In this section we present the program module<sup>4</sup> that handles *input* of CRS rules and metaterms according to the notational conventions of section 2.6 as adapted

---

<sup>4</sup>The section is the result of typesetting the literate Haskell script “crsinput.lhs”.

for the system in the previous section. The module is constructed using the RATATOSK system of Mogensen (1993).<sup>5</sup> The reader will benefit from knowledge of parser generation technology (Aho, Sethi and Ullman 1986) to appreciate this section. We first present the parser specification which is the essential piece of code, including some bureaucratic auxiliaries that serve to parse the somewhat esoteric T<sub>E</sub>X-notation used; most of this is, however, handled by the following scanner specification.

**6.3.1 Code (CRS parser).** The CRS parser converts a list of tokens into the corresponding CRS data object. The actual parser is in the file `crs_parse.hs` (not reproduced here) is generated from the RATATOSK parser specification in the file `crs.gram`, included here:

```
--$Id: crs.gram,v 2.8 1996/02/07 19:48:49 kris Exp kris $*-hugs*-
-- CRS interpreter: Ratatosk parser specification.
-- Copyright © 1995-1996 Kristoffer Høgsbro Rose, all rights reserved.

-- LITTERATE PROGRAMMING WRAPPER.

Goal      → Junk Precious endmarker Goal      $ case (x2,x4) of {
                                     (Crs a,Crs b) → Crs(a++b);
                                     (Crs _,Junk)  → x2;
                                     -
                                     → Junk } $
    | Junk Precious                    $ x2 $
    | Junk                              $ Junk $

Junk      → $$ | Junk Anything $$

Anything  → endmarker $$ | ident $$ | arrow $$
    | lpar $$ | rpar $$ | lbra $$ | rbra $$ | lang $$ | rang $$
    | comma $$ | semi $$ | hat $$ | dot $$
    | numb $$ | sym $$ | binop $$ | var $$ | vect $$ | metavar $$
    | anythingelse $$

Precious  → startcrs CRS                    $ Crs x2 $
    | startmetaterm MetaTerm                $ MetaTerm (satmv x2) $

-- CRS REWRITE SYSTEMS.

CRS      → Rules                            $ x1 $
```

---

<sup>5</sup>With a few trivial modifications to make the system adhere to Haskell syntax.

```

Rules      → startcrs Rules          $ x2 $
           | Rule                    $ [x1] $
           |      semi                $ [] $
           | Rule semi                $ [x1] $
           | Rule semi Rules          $ x1 : x3 $
           |      semi Rules          $ x2 $

Rule       → MetaTerm arrow MetaTerm ident $ satmv3(satmv x1,satmv x3,unbrace x4) $
           | MetaTerm arrow ident MetaTerm $ satmv3(satmv x1,satmv x4,unbrace x3) $
           | MetaTerm arrow MetaTerm      $ satmv3(satmv x1,satmv x3,rootname x1) $

-- CRS METATERMS.

MetaTerm   → lbra Vars rbra MetaTerm    $ foldr Mabs x4 x2 $
           | Seq                        $ x1 $

Seq        → App binop Seq              $ Mcon(preop(x2,2,True))[mkApp x1,x3] $
           | App dot MetaTerm           $ mkAbs x1 x3 $
           | App                         $ mkApp x1 $

App        → Simple App'                $ x1 : x2 $
           | var App'                   $ Mvar x1 : x2 $
           | M App''                     $ x1 : x2 $
           | S App''                     $ x1 : x2 $

App'       → Simple App'                $ x1 : x2 $
           | App''                       $ x1 $

App''      → var App'                   $ Mvar x1 : x2 $
           | M App''                     $ x1 : x2 $
           | S App''                     $ x1 : x2 $
           |                               $ [] $

Simple     → Symbol MetaTermList        $ Mcon (presym x1 x2) x2 $
           | MetaVar MetaTermList        $ Mapp x1 x2 $
           | MetaTermList                $ mkTuple x1 $
           | lang rang                    $ Mcon ("<>",0,False) [] $
           | lang MetaTermList' rang      $ Mcon ("<>",length x2,False) x2 $

M          → MetaVar                     $ Mapp x1 [] $
S          → Symbol                       $ Mcon (presym x1 []) [] $

MetaTermList → lpar rpar                 $ [] $
              | lpar MetaTermList' rpar  $ x2 $
              | lpar MetaTermList' rpar hat Sym

```

```

    $ [Mcon ("^",2,True) [mkTuple x2,Mcon(x5,0,False) []]] $
  | lpar MetaTermList' rpar hat metavar
    $ [Mcon ("^",2,True) [mkTuple x2,Mapp(premv(x5,0,False)) []]] $
  | lpar MetaTermList' rpar hat MetaTermList
    $ [Mcon ("^",2,True) [mkTuple x2,mkTuple x5]] $

MetaTermList' → MetaTerm          $ [x1] $
  | MetaTerm comma MetaTermList' $ x1 : x3 $

Vars      → var                    $ [x1] $
  | var Vars                        $ x1 : x2 $
  | var comma Vars                  $ x1 : x3 $
  | vect var numb                   $ mkVect x2 x3 $

MetaVar   → metavar hat numb       $ premv(x1,atoi(unbrace x3),False) $
  | metavar                          $ premv(x1,-1,False) $

Symbol    → Sym hat numb           $ (x1,atoi(unbrace x3),False) $
  | Sym                               $ (x1,-1,False) $

Sym       → sym                    $ x1 $
  | numb                              $ x1 $

$-- End of Ratatosk grammar.

-- Parse error function (generic one from Ratatosk package):

parse_error state input
  = error ("Parse_error_(state_++show state++)_in\n"
          ++case input of {[] → "END"; _ → echo_input input})

echo_input ((_,_,(line,s)):input)
  = "line_"++show line++"_at\n"
    ++s++case (lines (concat (map getstring input))) of
      { [] → ""; (h:t) → h }
  where getstring (_,_,(l,s)) = s

-- I/O declarations

data ParseTree = Crs CRS | MetaTerm Mterm | Junk

instance Text ParseTree where
  readsPrec d s = [(readParseTree s, "")]
  showsPrec d pt s = r++s
    where r = case pt of
      Crs c      → "CRS:\n"++showCRS c

```



```

        MetaTerm t → "METATERM:␣" ++ showMterm t
        Junk      → "Junk"

readParseTree :: String → ParseTree
readParseTree s = case crs_parse s of
    Crs c      → assert (checkCRS c) (Crs c)
    MetaTerm t → assert (checkMterm t) (MetaTerm t)
    Junk      → Junk

instance Text Mterm where
    readsPrec d s = [(readMterm s, "")]
    showsPrec d t s = showMterm t ++ s

readMterm :: String → Mterm
readMterm s = case crs_parse("METATERM:␣" ++ s) of
    MetaTerm t → assert (checkMterm t) t
    _          → error "Not␣a␣MetaTerm?"

$-- End of included Haskell.

```

As can be seen, the syntax is almost written as inductive definitions; the tokens (in lower case) are inherited from the scanner presented below. Quite a few forms of ‘syntactic sugar’ are allowed, notably all the abbreviations of Notation 2.6.5. For each production the ‘semantic action’ is given in \$ ... \$; they make use of several auxiliary functions presented below. The last declarations configure the parser’s error handling and declare standard Haskell input/output functions using the parser in combination with the printing routines of the previous section.

**6.3.2 Code (CRS parser auxiliaries).** The following functions are used as auxiliaries by the parser above.

Converting a string to an integer as usual.

```
1 atoi = foldl (λx y → 10*x + ord y - ord '0') 0
```

Building a tuple.

```
2 mkTuple [] = Mcon ("{}",0,False) []
3 mkTuple [t] = t
4 mkTuple ts = Mcon ("{}",length ts,False) ts
```

Building a curried application.

```
5 mkApp [] = error "PANIC---empty␣Application"
6 mkApp ts = foldl1 (λt1 t2 → Mcon ("@",2,False) [t1,t2]) ts
```

Building an 'F-abstraction'.

```

7 mkAbs [] _ = error "PANIC---empty_␣Abstraction"
8 mkAbs (Mcon (s,0,False) [] : []) _ =
9   error "PANIC---no_␣variables_␣in_␣Abstraction"
10 mkAbs (Mcon (s,0,False) [] : ts) t0 = mkAbs' (s,1,False) ts where
11   mkAbs' sym [] = t0
12   mkAbs' sym (Mvar v : ts) = Mcon sym [Mabs v (mkAbs' sym ts)]
13   mkAbs' sym _ =
14     error "PANIC---nonvariable_␣Abstraction_␣variable"
15 mkAbs _ _ = error "PANIC---bad_␣Abstraction"

```

Building a vector of similar symbols.

```

16 mkVect s n
17   = [ s++"_"++(if i<10 then show i else "{"++show i++"}")
18     | i ← [1..atoi(unbrace n)] ]

```

Extracting a name useable as tag from a symbol.

```

19 rootname (Mcon sym _) = [chr 36]++showSym sym++[chr 36]
20 rootname _ = "anonymous"

```

Checking that the explicit arity of a function symbol fits the actual arity.

```

21 presym (s,a,i) ts
22   | a ≡ (-1) = (s,length ts,i)
23   | a ≥ 0    = (s,a,i) -- consistency-checked later.

```

Special syntax for recognising saturated metavariables and invisible infix application.

```

24 preop ("{}",_ ,_) = ("@",2,False)
25 preop f = f
26 premv (s@(' \\' : 'm' : 'S' : _),a,_ ) = (s,-1,True)
27 premv (s@(' \\' : 'm' : 'V' : _),a,_ ) = (s,0,True)
28 premv m = m

```

Saturation is really an optimisation but it is needed for correct operation so we include it here.

```

29 satmv t = sm [] t where
30   sm vs t@(Mvar _) = t
31   sm vs (Mabs v t) = Mabs v (sm (v:vs) t)
32   sm vs (Mcon f ts) = Mcon f [ sm vs t | t ← ts ]
33   sm vs (Mapp (s,0,True) []) = Mapp (s,0,True) []
34   sm vs (Mapp (s,_,True) []) = Mapp (s,length vs,True)(map Mvar(sort vs))
35   sm vs (Mapp m@(s,a,_ ) ts)

```

```

36 | a>0 ^ length vs ≡ a ^ all isvar ts = Mapp (s,a,True) ts
37 | otherwise = Mapp (s,length ts,False) [ sm vs t | t ← ts ]
38 where isvar (Mvar _) = True
39       isvar _      = False

```

When we are handling the saturated metavariables in the RHS, however, we should reuse those of the LHS or fail!

```

40 satmv3(lhs,rhs,nm) = (lhs,sm' rhs,nm) where
41   sm' t@(Mvar _)      = t
42   sm' (Mabs v t)      = Mabs v (sm' t)
43   sm' (Mcon f ts)     = Mcon f [ sm' t | t ← ts ]
44   sm' t@(Mapp m@(_,0,True) []) = t
45   sm' t@(Mapp m@(s,_,True) ts) = findit ms where
46     findit [] = t
47     findit (t'@(Mapp (s',_,_) _) : ms') | s ≡ s'      = t'
48                                           | otherwise = findit ms'
49   sm' (Mapp m ts)      = Mapp m [ sm' t | t ← ts ]

```

The list of metavariables in the lhs is extracted like this.

```

50 ms = ms' lhs where
51   ms' (Mvar _)      = []
52   ms' (Mabs v t)    = ms' t
53   ms' (Mcon f ts)   = concat [ ms' t | t ← ts ]
54   ms' t@(Mapp (s,_,True) ts) = [t]
55   ms' (Mapp _ ts)   = concat [ ms' t | t ← ts ]

```

Finally, discard the  $\TeX$ / $\LaTeX$  markup used for tags.

```

56 unbrace ('\\':t':h':e':t':a':g':':rest) = init rest
57 unbrace ('\\':t':a':g':':rest) = init rest
58 unbrace ('T':A':G':':rest) = init rest
59 unbrace ('{':_':':rest) = init (init rest)
60 unbrace ('{':_':rest) = init rest
61 unbrace ('{':rest) = init rest
62 unbrace simple = simple

```

**6.3.3 Code (CRS tokens).** The CRS ‘scanner’ (that converts a string into a list of tokens to be parsed below) in the file `crs_scan.hs` (not reproduced here) is generated from the following RATATOSK scanner specification in the file `crs.tokens`.

```
--$Id: crs.tokens,v 2.7 1996/01/31 07:24:00 kris Exp kris $--hugs--
```

```
-- CRS interpreter: Ratatosk scanner specification.
```

```
-- Copyright © 1995-1996 Kristoffer Høgsbro Rose, all rights reserved.
```

```
-- TERMINALS. Note: update 'Anything' in crs.gram when the set changes.
```

```
startcrs      = "CRS:" ;
startmetaterm = "METATERM:" ;
endmarker     = "END" ;

ident = tag braced | "{_" braced '}' ;

arrow = "->" | "\\to" | "←" ;
lpar  = '(' | "\\(" | "\\<" | "\\[" | "{" ;
rpar  = ')' | "\\)" | "\\>" | "\\]" | "}" ) (@| '_' index ) ;
lbra  = '[' ;
rbra  = ']' ;
lang  = '<' ;
rang  = '>' ;
comma = ',' ;
semi  = ';' | "\\\\" (@| '[' nobrack* ']' ) ;
hat   = '^' ;
dot   = '.' ;

numb  = digit | bracednumber ;

var   = lowerletter (@| '_' index ) '\\'* ;

vect  = "\\$" ;

metavar = upperxyz (@| '_' index ) '\\'* ;

binop = op (@| '_' index ) '\\'* ;

sym   = other (@| '_' index ) '\\'* ;

anythingelse = ['\1'-'\8'] | '\11' | ['\14'-'\31'] | '!' | '#' | '$'
              | ['\''-'}] | ['\128'-'\255'] ;

-- COMMENTS.

Comment "%%" ( ['\1'-'\9'] | ['\14'-'\255'] )* '\n'
Comment "--" ( ['\1'-'\9'] | ['\14'-'\255'] )* '\n'
Comment "\\noindent " ( ['\1'-'\9'] | ['\14'-'\255'] )* '\n'
Comment "\\item" ( ['\1'-'\9'] | ['\14'-'\255'] )* '\n'
Comment "\\relax" ( ['\1'-'\9'] | ['\14'-'\255'] )* '\n'
Comment "\\text{" nobrace* "}"
Comment "\\quad"
Comment "\\qqquad"
Comment "\\intertext{" ( braced | nobrace | '\n' )* "}"
```

```

Comment "\\paragraph*" ( braced | nobrace | '\n' )* ""
Comment '&'
Comment "\\,"
Comment "\\begin{" nobrace* ""
Comment "\\end{" nobrace* ""
Comment '%'
Comment '~'
Comment '\\"

```

where

```

lower = ['a'-'z'] ;
upper = ['A'-'Z'] ;
letter = lower | upper ;
digit = ['0'-'9'] ;

```

-- INTERNALS.

```

upperxyz = ['U'-'Z'] | "\\m{" nobrace* '}' | "\\m" upper '{' nobrace* '}' ;

lowerletter = lower | "\\v{" nobrace* '}' ;

op = '*' | '+' | '-' | '|' | ':' | '=' | '/' | qop | "{" ;

qop = '' ('U'|'V'|'O'|'*'|'+'|'.'|'~'|'|'|':'|='|'/|';|'^')
      | "" qord qord*
      | '' braced | "[" nobrace* ']' | "(" nobrace* ')' ;

qord = '<' | '>' | '(' | ')' | '[' | ']' | '/' | '=' ;

other = qsym | boldqsym | braced | escaped | word
        | ['A'-'T'] | '!' | '?' | '@' | "\\c{" nobrace* '}' ;

qsym = '' ( letter | digit | '_' | '!' | '?' | '<' noangle* '>' ) ;
boldqsym = "@" ( letter | digit | qsym ) ;

tag = "\\tag" | "\\thetag" | "TAG" ;

braced = '{' nobrace nobrace* '}' ;

bracednumber = '{' digit digit* '}' ;

escaped = '\\ ' letter letter* ;

word = '\\ ('\' | '' ) braced ;

index = letter | digit | braced | escaped ;

```

```

nobracket = ['!' '-' 'Z'] | '\\\' | ['_ '-' '\255'] ;
noangle   = ['!' '-' ';' ] | '=' | ['?' '-' '\255'] ;
nopar     = ['!' '-' '\''] | ['*' '-' '\255'] ;

-- END of Ratatosk scanner.

```

Notice how the tokens are expected to be written using  $\text{\TeX}$  mathematics conventions (Knuth 1984): this is such that the examples of this dissertation can be run through the system literally.

## 6.4 Reduction

In this section we present the program module<sup>6</sup> with the functions for performing reduction in CRSs, following section 2.6’s Definition 2.6.8 through Definition 2.6.12 (and hence Klop, van Oostrom and van Raamsdonk 1993) as rigorously as possible, however, we have included some of the extensions described in this dissertation, notably

- the special *free variable matching constraint* on zero-ary metavariables of Remark 5.1.7,
- *addressing* written as  $(t)^a$  with natural numbers as addresses,
- an *address oracle* which means that the special construction  $t^{\text{fresh}}$  in RHSs is given a fresh address unique for the reduction step in question during substitution, and
- *sharing* which means that if addressed subterms are changed by a rewrite then all other subterms with the same address are also changed. This does *not* ensure the wfa property, of course: for that to hold a strategy that is ‘sharing safe’ for the rewrite system must be employed.
- *updating* through the special construction  $s[]t^a$  which is replaced with  $s[t^a]$  by the substitution operation, thus it has effect when it occurs in RHSs of rules.

(insofar as these things have affected the input and output form they are of course already present in earlier sections).

---

<sup>6</sup>The section is the result of typesetting the literate Haskell script “crsred.lhs”.

**6.4.1 Code (matching).** Determining whether the LHS ('pattern') of a particular rule matches a particular subterm ('occurrence') amounts to building a valuation as in Definition 2.6.8 but allowing for failure.

```
1 data Valuation = Valuation [(Sym, [Var], Mterm)]
2                 | MatchFail
```

Printing a valuation merely prints the components in an orderly way.

```
3 instance Text Valuation where
4   showsPrec d MatchFail rest      = "{Failure}" ++ rest
5   showsPrec d (Valuation []) rest = "\\{\\}" ++ rest
6   showsPrec d (Valuation vl) rest = "\\{"
7     ++ foldr1 (\s1 s2 → s1 ++ ";" ++ s2)
8     (map (\(s, vs, t) → showSym s ++ "\"->\"" ++ show (foldr Mabs t vs))
9         vl)
10    ++ "\\}" ++ rest
```

Constructing the valuation (or discovering that the match isn't possible) for a particular rule is defined using a match utility function (which we will have other uses for later). Since the valuation is to be used for rewriting, we insist on *safeness* (cf. Definition 2.6.9) of it. Matching is a generic inductive definition with an 'accumulator' for the component substitutes.

```
11 match :: Mterm → Mterm → Valuation
12 match lhs t = match' (fresh []) lhs (Valuation []) t
```

In order to facilitate unification (to be defined later) we allow an initial valuation to be passed into the match primitive valuation construction.

```
13 match' :: [Var] → Mterm → Valuation → Mterm → Valuation
14 match' contextvs lhs vl t =
15   mat lhs t notrans (\vl vs → vl) vl safevs where
```

First we establish the safevs which are just those variables that are neither bound in the context (cf. Code 6.4.2 below) nor in the pattern.

```
16   safevs = filter ('notElem' (bv lhs)) contextvs
```

We 'accumulate' using the following (locally defined) mat which is in continuation-passing style to allow quick exit on failure.

```
17   mat :: Mterm → Mterm → Trans Var → (Valuation → [Var] → Valuation)
18       → Valuation → [Var] → Valuation
```

A variable in the pattern should correspond to the right variable in the occurrence.

```

19 mat (Mvar v) (Mvar v') vts c vl safevs
20   | trans vts v ≡ v'      = c vl safevs
21   | otherwise             = MatchFail

```

Conversely, an abstraction matches if the body matches modulo the implicit renaming (this is the only place the variable translation is extended).

```

22 mat (Mabs v t) (Mabs v' t') vts c vl safevs
23     = mat t t' (xtrans v v' vts) c vl safevs

```

And a construction is a straightforward ripple.

```

24 mat (Mcon f ts) (Mcon f' ts') vts c vl safevs
25   | f ≡ f'              = mats' ts ts' c vl safevs
26   | otherwise           = MatchFail
27   where mats' [] [] c = c
28         mats' (t:ts) (t':ts') c = mat t t' vts (mats' ts ts' c)
29         mats' _ _ c = (λvl vs → MatchFail)

```

The special *free variable matching constraint* on metavariables is treated before the general case.

```

30 mat (Mapp m@(_,0,True) []) t' vts c (Valuation vl) safevs
31   | isfv t'              = c (Valuation ((m,[],t') : vl)) safevs
32   | otherwise            = MatchFail
33   where isfv (Mvar v') | istrans vts v' = False
34                   | otherwise          = True
35         isfv _                = False

```

Only the following last case is interesting since it is when the pattern contains a metaapplication that we add a substitute to the valuation. Notice that this is the only place where we actually manipulate the internals of the valuation and consume variables from the list of safe variables (the safe variables are those neither in the context, as passed to match initially, nor in the rule). There are two ways a match can succeed:

- A. For a repeated metavariable we must ensure that the substitute is also valid for the this subterm (modulo renaming to the possibly different variables).
- B. For a new metavariable we must ensure that the suboccurrence that is to become the body of the valuation only contains the explicitly allowed (free) variables – this is particularly easy for *saturated* metaapplications, of course.

Here are the success cases and the failure case.





```

54 rebind :: Trans Var → Mterm → [Var] → (Mterm, [Var])
55 rebind tr (Mvar v)    vs      = (Mvar (trans tr v), vs)
56 rebind tr (Mabs v t) [] = error "PANIC--empty_infinite_fresh_variable_list"
57 rebind tr (Mabs v t) (v':vs') = (Mabs v' t'', vs'')
58   where (t'',vs'') = rebind (xtrans v v' tr) t vs'
59 rebind tr (Mcon f ts) vs      = (Mcon f ts', vs')
60   where (ts',vs') = rebind' tr ts vs
61 rebind tr (Mapp m ts) vs      = (Mapp m ts', vs')
62   where (ts',vs') = rebind' tr ts vs
63 rebind' _ []    vs = ([],vs)
64 rebind' tr (t:ts) vs = (t':ts'',vs'')
65   where (t',vs')    = rebind tr t vs
66           (ts'',vs'') = rebind' tr ts vs'

```

Finally the generic definition of bound variables is the following.

```

67 bv :: Mterm → [Var]
68 bv (Mvar v)    = []
69 bv (Mabs v t)  = v : bv t
70 bv (Mcon f ts) = concat [ bv t | t ← ts ]
71 bv (Mapp m ts) = concat [ bv t | t ← ts ]

```

**6.4.2 Code (redex).** We provide a means to search a term for all redexes as defined in Definition 2.6.12.A. For efficiency the corresponding valuation and rule are incorporated into the Redex datatype such that it is later easier to rewrite it.

```
1 data Redex = Rx Path Valuation Rule
```

Redexes are ordered as their paths, *i.e.*, from left to right (this is useful for some reduction strategies).

```

2 instance Eq Redex where
3   (Rx p _ (_,_,r)) ≡ (Rx p' _ (_,_,r')) = p ≡ p' ∧ r ≡ r'
4 instance Ord Redex where
5   (Rx p _ _) ≤ (Rx p' _ _) = p ≤ p'
6 instance Text Redex where
7   showsPrec d (Rx p vl (lhs,_,nm)) s =
8     "{" ++ concat (map shownumb p) ++ ":" ++ nm ++ "}" ++ s

```

A path describes the occurrence of the hole in the context  $C[]$  of Definition 2.6.12.A as a list of  $!!$ -indices.<sup>7</sup>

<sup>7</sup>Note that Haskell indexes lists starting from 0.

```
9 type Path = [Int]
```

All redexes can be collected at once; this is also where we build the initial list of free variables such that the valuations will all use different variables.

```
10 redexes :: CRS → Mterm → [Redex]
11 redexes crs t = red [] t where
12   red :: Path → Mterm → [Redex]
13   red _ (Mvar _) = []
14   red p (Mabs v t) = red (p++[0]) t
15   red p t@(Mcon f ts) =
16     [ Rx p v1 r | r ← crs, v1 ← matches r ]
17     ++concat [ red (p++[n]) t | (t,n) ← zip ts [0..] ]
18   where matches (lhs,rhs,nm) =
19         case match' (fresh usedvs) lhs (Valuation[]) t of
20           MatchFail → []
21           v1 → [v1]
22   red p t@(Mapp _ ts) =
23     concat [ red (p++[n]) t | (t,n) ← zip ts [0..] ]
24   usedvs = nub (bv t++fv t)
```

Extraction of subterms useful in general and thus made global.

```
25 subterm :: Path → Mterm → Mterm
26 subterm [] t = t
27 subterm _ (Mvar v) = error "PANIC---illegal_Path"
28 subterm (i:p) (Mabs v t) | i ≡ 0 = subterm p t
29 | otherwise = error "PANIC---illegal_Path"
30 subterm (i:p) (Mcon s ts) | 0 ≤ i ∧ i < length ts = subterm p (ts !! i)
31 | otherwise = error "PANIC---illegal_Path"
32 subterm (i:p) (Mapp m ts) | 0 ≤ i ∧ i < length ts = subterm p (ts !! i)
33 | otherwise = error "PANIC---illegal_Path"
```

**6.4.3 Code (substitution).** Substitution is the simple art of applying the homomorphic extension of a valuation to a metaterm; this is easy because of the safeness.

```
1 type Subst = Mterm → Mterm
```

The implementation is straightforward since we know the applied valuation to be safe provided we rebind variables to the supplied fresh ones.

We carefully maintain the *addresses* of section 5.2 during substitution by replacing the special address <sup>fresh</sup> with the reduction number (a convenient oracle).

```

2 subst :: Int → [Var] → Valuation → Subst
3 subst _ _ MatchFail = error "PANIC---illegal substitution"
4 subst n fvs (Valuation vl) = sub fvs [] where
5   sub :: [Var] → [(Var,Mterm)] → Subst
6   sub fvs vts (Mvar v) = (transt vts v) where
7     transt [] v = Mvar v
8     transt ((v',t'):vts) v | v ≡ v' = t'
9                             | otherwise = transt vts v
10  sub (v':fvs') vts (Mabs v t) = Mabs v' (sub fvs' ((v,Mvar v'):vts) t)
11  sub fvs vts (Mcon f ts) = t' where
12    t' | f ≡ freshfs = Mcon (shownumb n,0,False) []
13        | f ≡ updatefs = update (sub fvs vts (head ts))
14                               (map (sub fvs vts) (tail ts))
15        | otherwise = Mcon f [ sub fvs vts t | t ← ts ]
16  sub fvs vts (Mapp m ts) =
17    sub fvs (zip vts' [ sub fvs vts t | t ← ts ]) t' where
18      (vts',t') = mfind vl
19      mfind ((m',vts',t'):ms) | m ≡ m' = (vts',unmeta t')
20                              | otherwise = mfind ms
21      mfind [] = error("PANIC---metavariable_" ++ showSym m ++ "_not_found")
22  freshfs = ("\\fresh",0,False)
23  updatefs = ("'{update}'",2,True)

```

**6.4.4 Code (rewrite step).** A rewrite is a contraction of a designated set of redexes (rather than of a single redex as in Definition 2.6.12.C).

```

1 rewrite :: Int → [Redex] → Mterm → Mterm

```

The redexes are contracted innermost-first (backwards) – hence `rewrite` fails if there are overlapping redexes; also the oracle argument to `subst` *remains the same throughout the rewrite*.

```

2 rewrite n rxs t = tr (show n+"/" ++ show(length rxs)
3                    ++ concat["_" ++ show rx|rx ← rxs]) t' where
4   t' = remeta (rew rxs)
5   rew [] = unmeta t

```

```

6   rew (rx:rxs) = rew' rx (rew rxs)
7   rew' (Rx p vl (lhs,rhs,nm)) t = update t' us' where
8     (t',us') = rew'' [] p t
9     rew'' vs [] t = (subst n (fresh vs) vl rhs,[])
10    rew'' vs ( _:_) (Mvar _) = nopath
11    rew'' vs (i:p) (Mabs v t) | i ≡ 0 = (Mabs v t',us')
12                                | otherwise = nopath
13                                where (t',us') = rew''(v:vs) p t
14    rew'' vs (i:p) t@(Mcon f ts)
15      | f ≡ ("^",2,True) = (t',t':us')
16      | otherwise = (t',us')
17      where t' = Mcon f ts'
18            (ts',us') = rew''' vs p i ts
19    rew'' vs (i:p) (Mapp m ts) = (Mapp m ts',us')
20      where (ts',us') = rew''' vs p i ts
21    rew''' vs _ _ [] = nopath
22    rew''' vs p 0 (t:ts) = (t':ts,us') where (t',us') = rew'' vs p t
23    rew''' vs p i (t:ts) = (t:ts',us') where (ts',us') = rew''' vs p (i-1) ts
24    nopath = error "PANIC---impossible_Path"

```

Metarewriting is supported by simply lifting metavariables to (otherwise unique) constructors before substitution (and in substituted bodies) using a simple encoding of metavariables as function symbols and an associated decoding.<sup>8</sup>

```

25 unmeta t@(Mvar _) = t
26 unmeta (Mabs v t) = Mabs v (unmeta t)
27 unmeta (Mcon f ts)
28   | iscodedmetavar f = Mcon (metaencode f) [ unmeta t | t ← ts ]
29   | otherwise = Mcon f [ unmeta t | t ← ts ]
30 unmeta (Mapp m ts) = Mcon (metaencode m) [ unmeta t | t ← ts ]
31 remeta t@(Mvar _) = t
32 remeta (Mabs v t) = Mabs v (remeta t)
33 remeta (Mcon f ts)
34   | iscodedmetavar f = Mapp (metadecode f) [ remeta t | t ← ts ]
35   | otherwise = Mcon f [ remeta t | t ← ts ]
36 remeta (Mapp m ts) = Mapp m [ remeta t | t ← ts ]

```

The coding is an obscure misuse of the notation: a <sup>M</sup> is added in front.

```

37 iscodedmetavar ('{':'^': 'M':rest,a,i) = True

```

<sup>8</sup>We are thankful to Vincent van Oostrom for suggesting this trick.

```

38 iscodedmetavar _ = False
39 metaencode ('':s,a,i) = ("^M." ++ s,a,i)
40 metaencode (s,a,i) = ("^M" ++ s ++ ")" ,a,i)
41 metadecode ('':'^':'M':'.':s,a,i) = ('':s,a,i)
42 metadecode ('':'^':'M':s,a,i) = (init s,a,i)
43 metadecode f =
44   error("PANIC---decoding_␣nonmetavariable_␣" ++ showSym f)

```

Here the addressing has effect in that rewriting is *shared*: for each rewrite we keep track of the innermost address affected and update the associated term.

```

45 update :: Mterm → [Mterm] → Mterm
46 update t [] = t
47 update t us = (upd t (concat (map flatten us))) where
48   flatten (Mvar v) = []
49   flatten (Mabs _ t) = flatten t
50   flatten u@(Mcon f ts)
51     | f ≡ ("^",2,True) = [u]
52     | otherwise = concat [ flatten t | t ← ts ]
53   flatten (Mapp _ ts) = concat [ flatten t | t ← ts ]
54   upd t [] = t
55   upd t (u:us) = upd (upd1 t u) us
56   upd1 t@(Mvar v) u = t
57   upd1 (Mabs v t) u = Mabs v (upd1 t u)
58   upd1 (Mcon f@("^",2,True) [t,a]) u@(Mcon _ [_ ,a'])
59     | a ≡ a' = u
60     | otherwise = Mcon f [upd1 t u,a]
61   upd1 (Mcon f ts) u = Mcon f [ upd1 t u | t ← ts ]
62   upd1 (Mapp m ts) u = Mapp m [ upd1 t u | t ← ts ]

```

Finally we remark that unrestricted reduction relation as such is not readily programmable – that is why we devote the following section to reduction using a *strategy*.

## 6.5 Strategies

In this section we present the program module<sup>9</sup> with the functions for strategies for CRSs.

<sup>9</sup>The section is the result of typesetting the literate Haskell script “crsstrat.lhs”.

**6.5.1 Code (reduction strategies).** A reduction strategy selects for any term some of the redexes to be reduced, and maybe changes the term

```
1 type Strategy = Mterm → [Redex] → [Redex]
```

Strategies are sometimes a combination (intersection) of partial strategies.

```
2 comb :: Strategy → Strategy → Strategy
```

```
3 comb s1 s2 = λt rxs → s2 t (s1 t rxs)
```

A metaterm is irreducible if the reduction strategy returns the empty list; deterministic ones return lists of length at most 1.

**6.5.2 Code (standard reduction strategies).** Here we give some constant strategies that are often useful.

First the ‘leftmost’ and ‘rightmost’ strategy components which can exploit that redexes are ordered lexicographically as their paths.

```
1 leftmost, rightmost :: Strategy
```

```
2 leftmost _ [] = []
```

```
3 leftmost _ rxs = [head rxs]
```

```
4 rightmost _ [] = []
```

```
5 rightmost _ rxs = [last rxs]
```

Next the ‘outermost’ and ‘innermost’ strategies that remove all redexes that are inside the previous and outside the following redex, respectively.

```
6 outermost, innermost :: Strategy
```

```
7 outermost _ [] = []
```

```
8 outermost _ (rx:rxs) = rx : om' rx rxs where
```

```
9   om' :: Redex → [Redex] → [Redex]
```

```
10   om' _ [] = []
```

```
11   om' rx' (rx:rxs) | redexprefix rx' rx = om' rx' rxs
```

```
12                       | otherwise = rx : om' rx rxs
```

```
13 innermost _ [] = []
```

```
14 innermost _ (rx:rxs) = im' rx rxs where
```

```
15   im' rx' [] = [rx']
```

```
16   im' rx' (rx:rxs) | redexprefix rx' rx = im' rx rxs
```

```
17                       | otherwise = rx' : im' rx rxs
```

Both rely on the redexprefix predicate on paths.

```
18 redexprefix :: Redex → Redex → Bool
```

```

19 redexprefix (Rx p' _ _ ) (Rx p _ _ ) = isprefix p' p
20 isprefix [] _           = True
21 isprefix _ []          = False
22 isprefix (n':ns') (n:ns) | n ≡ n'      = isprefix ns' ns
23                               | otherwise = False

```

The ‘weak’ strategy filters out the redexes inside an abstraction; the ‘strong’ strategy removes nothing.

```

24 weak, strong :: Strategy
25 weak t rxs = filter isweak rxs where
26   isweak (Rx p _ _ ) = isweak' t p
27   isweak' _          [] = True
28   isweak' (Mabs v t) _ = False
29   isweak' (Mconf ts) (n:p) = isweak' (ts !! n) p
30   isweak' (Mapp m ts) (n:p) = isweak' (ts !! n) p
31 strong _ rxs = rxs

```

Finally, the ‘flat’ strategy.

```

32 flat :: Strategy
33 flat _ rxs = filter (λ(Rx p _ _ ) → p ≡ []) rxs

```

Next the support for ‘repeated’ reduction. This always involves a primary and secondary CRS wher the idea is that for each step of the primary CRS the secondary CRS is applied completely (using `nf` with the outermost strategy above so the secondary CRS better be confluent and normalising). In most cases the secondary CRS will be empty, of course.

**6.5.3 Code (standard reduction).** Reducing with some strategy a single step or until normal form means just that. Only good for deterministic nonoverlapping strategies.

```

1 onestep, nf :: CRS → CRS → Strategy → Mterm → Mterm
2 onestep crs crs2 strat t = rewrite 0 (strat t (redexes crs t)) t

```

Reduction to normal form simply repeats a single rewrite step (of the primary CRS) until there are no more redexes.

```

3 nf [] [] _ t = t
4 nf crs crs2 strat t = nf' 0 t where

```



All the work is done by the internal `nf'` that and does a rewrite, tracing the reduction count.

```

5   nf' n t | rxs ≡ [] = t'
6       | otherwise = nf' n1 (rewrite n1 rxs t')
7       where rxs = strat t' (redexes crs t')
8             n1 = n+1
9             t' = nf crs2 [] outermost t

```

The `nf` function verifies the WN property for a term.

**6.5.4 Code (exhaustive reduction).** This is the opposite: here we search the entire space of possible reductions allowed by the strategy, returning the *list* of normal forms. This is always safe.

```

1  nfs :: CRS → CRS → Strategy → Mterm → [Mterm]
2  nfs [] [] _ t = [t]
3  nfs crs crs2 strat t = nfs' 0 [] [t] where
4  nfs' :: Int → [Mterm] → [Mterm] → [Mterm]

```

When no more terms need reducing we are done.

```

5  nfs' n _ [] = []

```

Otherwise we 'output' the list of normal forms lazily by returning any of the current terms already in normal form and proceeding with reducing the rest.

```

6  nfs' n old current = tr (show n++"."++show (length new))
7                        (newnfs++nfs' n1 (old++newnfs) new)
8  where
9  n1 = n+1

```

All new normal forms are extracted into `newnfs`, and the remaining terms with all their redexes reduced are collected in `new`.

```

10 newnfs = [ t | (t,_) ← nfp, t 'notElem' old ]
11 new = [ rewrite n [rx] t | (t,rxs) ← nonnfp, rx ← rxs ]

```

These are all based on a partitioning of `current` into the normal forms and the redexes.

```

12 (nfp,nonnfp) =
13   partition (λ(_,rx) → rx ≡ [])
14   [ (t, strat t (redexes crs t)) | t ← current' ]
15 current' = map (nf crs2 [] outermost) (nub current)

```

The `nfs` function can be said to verify the SN property for a term.

## 6.6 Analysis

In this section we present the program module<sup>10</sup> with the functions for performing analysis of CRSs, following section 2.6's Definition 2.6.14 and several definitions of chapter 5.

**6.6.1 Code (left-linearity).** We encode Definition 2.6.14.A.

```

1 leftlinear :: CRS → Check
2 leftlinear crs = either "CRS_not_leftlinear:" (ll' crs) where
3   ll' [] = ""
4   ll' ((lhs,_,nm):rs) =
5     either ("\n_in_rule_" ++ nm ++ ":") (linear lhs) ++ ll' rs

```

The encoding of the linearity property for terms exploits that our 'sets' are really lists of occurrences.

```

6 linear :: Mterm → Check
7 linear t = case [ showSym m | m ← duplicates (mv t) ] of
8   [] → ""
9   ms → "duplicated_metavariables:"
10      ++ concat [" " ++ m | m ← ms] ++ "."

```

**6.6.2 Code (nonoverlapping).** We encode Definition 2.6.14.B. An overlap is represented as a metaterm with two redexes.

```

1 type Overlap = (Mterm,Redex,Redex)

```

The check merely announces all the critical pairs in a readable way.

```

2 nonoverlapping,weaklynonoverlapping :: CRS → Check
3 nonoverlapping crs =
4   either "CRS_has_overlaps:" (showOverlaps (overlaps crs))
5 showOverlaps [] = ""
6 showOverlaps ((t,rx1,rx2):os) =
7   "\nterm_" ++ show t ++ "\nmatches_" ++ show rx1 ++ "\n_and_" ++ show rx2

```

Weakly overlapping means critical pairs trivial.

---

<sup>10</sup>The section is the result of typesetting the literate Haskell script "crsanal.lhs".

```

8 weaklynonoverlapping crs =
9   either "CRS_has_overlaps:"
10         (showOverlaps (filter dropequalpairs (overlaps crs))) where
11     dropequalpairs (t,rx1,rx2) =
12     rewrite 0 [rx1] t # rewrite 0 [rx2] t

```

The overlaps of a CRS are all smallest terms with two overlapping redexes.

```

13 overlaps :: CRS → [Overlap]
14 overlaps crs = cp' crs where
15   cp' [] = []
16   cp' (r:rs) = overlaps' r r ++ cp1 r rs ++ cp' rs
17   cp1 r [] = []
18   cp1 r (r':rs') = overlaps' r r' ++ cp1 r rs'

```

Actually constructing the rule overlaps involves first searching for candidates (where the root symbol of one rule occurs in the other) followed by a check whether the combined superposition of the terms is, in fact, an overlap.

```

19 overlaps' :: Rule → Rule → [Overlap]

```

If the two rule arguments are the same then self-overlaps and mirror overlaps are omitted.

```

20 overlaps' r1@(t1@(Mcon s1 t1s),_,_) r2@(Mcon s2 t2s,_,_) =
21   (if r1 ≡ r2 then [] else concat [ try r1 r2 p | p ← pw s1 t1 ])
22   ++ concat [ try r2 r1 (i:p) | (i,t2) ← zip [0..] t2s, p ← pw s1 t2 ]
23   where

```

The following function generates a list of potential overlap occurrences in the form of subterms with a particular symbol at the root.

```

24   pw :: Sym → Mterm → [Path]
25   pw s (Mvar _) = []
26   pw s (Mabs _ t) = map (0:) (pw s t)
27   pw s (Mcon s' ts) | s ≡ s' = [] : rest
28                       | otherwise = rest
29     where rest = [(i:p) | (i,t) ← zip [0..] ts, p ← pw s t]
30   pw s (Mapp _ _) = []

```

Testing whether a particular occurrence is an overlap is merely attempting to construct the two valuations from the overlaid term.

```

31   try r@(t,_,_) r'@(t',_,_) p =

```

```

32     case match t st of
33       MatchFail → []
34       vl        → case match t' (subterm p st) of
35                   MatchFail → []
36                   vl'       → [(st,Rx [] vl r,Rx p vl' r')]
37     where st = overlay t p t'

```

**6.6.3 Code (unification).** We will implement a simple kind of unification: an ‘overlay’ is a copy of  $t$  except the subterm at an occurrence  $p$  has been overlaid with  $t'$  with any metavariables in the latter renamed to be distinct from those of  $t$ . This is a generalisation of matching in that the two patterns are checked to be identical until a metavariable is encountered.

```

1  overlay :: Mterm → Path → Mterm → Mterm
2  overlay t p t'
3    | check(linear t++linear t') = ov t p []
4    | otherwise = error"LIMITATION---overlay_only_works_for_linear_patterns"
5  where

```

The easy part is the part of  $t$  which is unchanged, except we have to keep track of the variables defined.

```

6  ov :: Mterm → Path → [Var] → Mterm
7  ov (Mvar v)    (i:p) vs = failureterm [v]
8  ov (Mabs v t)  (i:p) vs = Mabs v (ov t p (v:vs))
9  ov (Mcon s ts) (i:p) vs =
10   Mcon s (take i ts++[ov (ts!!i) p vs]++drop (i+1) ts)
11  ov (Mapp m ts) (i:p) vs =
12   Mapp m (take i ts++[ov (ts!!i) p vs]++drop (i+1) ts)
13  ov t          []      vs = ov' vs (mktrans vs vs) t t'

```

Once we have reached the point where real insertion happens things get more interesting. While no metavariables are involved we merely check that the terms are identical and overlay subterms.

```

14  ov' :: [Var] → Trans Var → Mterm → Mterm → Mterm
15  ov' vs vt t1@(Mvar v1) (Mvar v2)
16    | v1 ≡ trans vt v2                = t1
17    | otherwise                        = failureterm [v1]
18  ov' vs vt (Mabs v1 t1) (Mabs v2 t2) =

```

```

19     Mabs v1 (ov' vs (xtrans v2 v1 vt) t1 t2)
20   ov' vs vt (Mcon s1 t1s) (Mcon s2 t2s)
21     | s1 ≡ s2                               = Mcon s1 o12s
22     | otherwise                             = failureterm []
23   where o12s = zipWith (ov' vs vt) t1s t2s

```

Next we treat the special cases where either of the sides is one of the special variable-only matching metavariables.

```

24   ov' vs vt t1 (Mapp m2@(_,0,True) []) =
25     case t1 of Mvar v1 | istrans vt v1 → t1
26                 | otherwise → failureterm []
27                 - → failureterm []
28   ov' vs vt (Mapp m1@(_,0,True) []) t2 =
29     case t2 of Mvar v2 | hastrans vt v2 → Mvar (trans vt v2)
30                 | otherwise → failureterm []
31                 - → failureterm []

```

The last two cases are the only non-trivial ones: when one or both sides are metaapplications. The easy direction is when the metaapplication is in the inserted term (so no renaming metavariables is needed). We still have to ‘shave off’ the variables that are not permitted.

```

32   ov' vs vt t1 (Mapp m2 t2s) = shave vs vt t2s t1

```

we check the consistency of the created term while creating it (such that matching will fail quickly; this could be optimised in the same way as we did for matching in Code 6.4.1, of course, but we will not use this for computation so the efficiency is not critical here).

The hard direction has the metaapplication in the original term: we now have to insert the entire remainder of the inserted term where we also have to rename variables and even metavariables that occur in the original term.

```

33   ov' vs vt (Mapp m1 t1s) t2 = shave [] notrans t1s (renx freshvs vt t2)
34   where
35     renx fvs vt (Mvar v) = Mvar (trans vt v)
36     renx (v':fvs) vt (Mabs v t) =
37       Mabs v' (renx fvs (xtrans v v' vt) t)
38     renx fvs vt (Mcon f ts) = Mcon f (map (renx fvs vt) ts)
39     renx fvs vt (Mapp (s,a,f) ts) =
40       Mapp (trans trmv s,a+length newvars,f)
41         (map (Mvar) (oldvars++(vs'but'oldvars)))

```

```

42     where oldvars = [trans vt v | (Mvar v) ← ts]
43           newvars = (nub vs) 'but' oldvars
44
45     freshvs = fresh (nub(vs++fv t++fv t'++bv t++bv t'))
46
47     trmv = mktrans s_both_t_and_t' s_neither_t_nor_t' where
48
49     s_t,s_t',s_both_t_and_t',s_neither_t_nor_t' :: [String]
50
51     s_t    = map (λ(s,_,_) → s) (nub (mv t))
52     s_t'  = map (λ(s,_,_) → s) (nub (mv t'))
53
54     s_both_t_and_t' = filter ('elem' s_t) s_t'
55
56     s_neither_t_nor_t' =
57       filter ('notElem' (s_t++s_t'))
58         ([ [v] | v ← "ZYX" ]
59          ++ [ [v] ++ "_" ++ shownumb n | v ← "ZYX", n ← [1..9] ]
60          ++ [ [v] ++ "_" ++ shownumb n | n ← [10..], v ← "ZYX" ])
61
62     ov' _ _ t1      t2      = failureterm (nub (fv t1))

```

Shaving means remove from metaapplications in the overlaid terms those variables that do not appear in 'the other component' which

```

56     shave vs vt ts t = shave' allowedvs t where
57
58     allowedvs = vs++vs_in_ts ts
59
60     vs_in_ts []           = []
61     vs_in_ts (Mvar v : ts) = trans vt v : vs_in_ts ts
62     vs_in_ts _ = error"PANIC---non-variable_in_pattern_metaapplication"
63
64     shave' vs (Mvar v)    | v 'elem' vs = Mvar v
65                         | otherwise   = failureterm [v]
66     shave' vs (Mabs v t) = Mabs v (shave' (v:vs) t)
67     shave' vs (Mcon f ts) = Mcon f [ shave' vs t | t ← ts ]
68     shave' vs (Mapp (s,a,i) ts) = Mapp (s,length ts',i) ts' where
69       ts' = extractvs ts
70
71     extractvs []           = []
72     extractvs (Mvar v : ts) | v 'elem' vs = Mvar v : extractvs ts
73                         | otherwise   = extractvs ts
74     extractvs _ = error"PANIC---non-variable_in_pattern_metaapplication"

```

Finally the unmatchable term we use above, which takes a variable list to have the 'right' free variables to prevent matching against it from succeeding!

```

71     failureterm vs = Mcon ("fail/overlay",length vs,False)
72                   [ Mvar v | v ← vs ]

```

#### 6.6.4 Code (orthogonality). Just Definition 2.6.14.C.

```

1 orthogonal, weaklyorthogonal :: CRS → Check
2 orthogonal crs =
3   either (leftlinear crs
4           + either "\n" (nonoverlapping crs)) "\nCRS ⊄ orthogonal."
5 weaklyorthogonal crs =
6   either (leftlinear crs
7           + either "\n" (weaklynonoverlapping crs))
8           "\nCRS ⊄ weakly ⊄ orthogonal."

```

**6.6.5 Code (critical pairs).** The critical pairs are merely the result of reducing each of the two redexes of an overlap. We build the critical pairs as a CRS such that it can later be used as such (presumably after reversing some of the pairs to choose the opposite direction, and cleaning out in the old rules to avoid creating overlaps).

```

1 criticalpairs :: CRS → CRS
2 criticalpairs crs = map stepboth (overlaps crs) where
3   stepboth (t, rx1@(Rx p1 _ (_,_, nm1)), rx2@(Rx p2 _ (_,_, nm2))) =
4     (rewrite 0 [rx1] t, rewrite 0 [rx2] t, nm1++"---"++nm2)

```

**6.6.6 Code (minimalise CRS).** To ‘minimalise’ a CRS is to remove all rules that are subsumed by other rules.

```

1 minimalise :: CRS → CRS
2 minimalise crs = min' [] crs where
3   min' _ [] = []
4   min' pre (r:post) | (pre++post) 'subsumes' [r] = min' pre post
5                       | otherwise = r : min' (r:pre) post

```

A CRS with the rules  $\{r_i : p_i \rightarrow t_i\}$  *subsumes* the CRS with the rules  $\{r'_j : p'_j \rightarrow t'_j\}$  (such that the relation defined by the first has the relation of the second as a subrelation) if for all  $r'_j$  there is some  $r_i$  such that  $p$  metamatches  $p'$  and furthermore rewriting  $p$  with  $r'$  yields  $t$ .

```

6 subsumes :: CRS → CRS → Bool
7 subsumes crs crs' = all subsumedbycrs crs' where
8   subsumedbycrs r' = any ('sub1'r') crs

```

```

9   sub1 :: Rule → Rule → Bool
10  sub1 r@(p,t,nm) r'@(p',t',nm') =
11    case match p p' of
12      MatchFail → False
13      v1 | t' ≡ rewrite 0 [Rx [] v1 r] t → True
14      | otherwise                               → False

```

**6.6.7 Code (reversible CRS).** A reversible CRS is a CRS where the system obtained by reverting all the arrows – the *converse* – is also a CRS.

```

1  reversible :: CRS → Check
2  reversible crs = either "CRS_not_reversible_(converse_not_CRs):"
3                      (checkCRS (converse crs))
4  converse :: CRS → CRS
5  converse crs = assert (checkCRS crs') crs' where
6    crs' = conv' crs
7    conv' [] = []
8    conv' ((lhs,rhs,nm):rs) = (rhs,lhs,nm):conv' rs

```

**6.6.8 Code (local confluence).** This procedure searches tests the normal forms of all critical pairs, hence will only terminate if this search terminates.

```

1  locallyconfluent :: CRS → Check
2  locallyconfluent crs =
3    either "CRS_not_locally_confluent:\n"
4          (unlines
5            [ show t ++ "_reduces_to_both_" ++ show t1 ++ "_and_" ++ show t2
6              | (t,rx1,rx2) ← overlaps crs,
7                t1 ← nfs crs [] strong (rewrite 0 [rx1] t),
8                t2 ← nfs crs [] strong (rewrite 0 [rx2] t),
9                t1 ≠ t2 ])

```

## 6.7 Explicification

This section contains the implementation of the *CRS explicification* procedure defined in section 5.1.<sup>11</sup>

For ease of reference we repeat the relevant portions of Figure 5.1 for each code section.

<sup>11</sup>The section is the result of typesetting the literate Haskell script “crsx.lhs”.



**6.7.1 Code (explicification).** The main explicification procedure takes a CRS into another CRS containing substitution introduction, distribution, and elimination rules, as described in Definition 5.1.9.

```
1 explicify :: CRS → CRS
```

```
2 explicify rules = explicify' "S" rules
```

The variant actually used allows customisation of the symbol used for explicit substitution constructions.

```
3 explicify' :: String → CRS → CRS
```

```
4 explicify' z rules =
```

```
5   nub (introductionrules
```

```
6       ++concat [ xistribute f s | f ← symbols, s ← sigmas z ]
```

```
7       ++concat [ xeliminate s | s ← sigmas z ])
```

```
8   where
```

```
9     symbols          = symbolsin rules
```

```
10    introductionrules = [xintroduction z r symbols | r ← rules]
```

```
11    sigmas s          = symbolsin introductionrules \\ symbols
```

```
12    symbolsin []      = []
```

```
13    symbolsin ((rhs,lhs,_) : rs) = nub(syms lhs ++ syms rhs ++ symbolsin rs)
```

```
14    where
```

```
15      syms (Mvar _)      = []
```

```
16      syms (Mabs v t)    = syms t
```

```
17      syms (Mcon fs ts)
```

```
18          | fs'elem'dummies = nub (concat [ syms t | t ← ts ])
```

```
19          | otherwise       = nub (fs : concat [ syms t | t ← ts ])
```

```
20      syms (Mapp _ ts) = nub (concat [ syms t | t ← ts ])
```

```
21      dummies = [{"^",2,True},("{update}",2,True),
```

```
22                "\\fresh",0,False},{"!",0,False}]
```

**6.7.2 Code (substitution introduction).** Construct the rules (r-x) by substituting for each non-explicit metaapplication  $Z^n(\vec{t}_{(n)})$  the explicit construction  $\Sigma^{n+1}([\vec{x}]Z^n(\vec{x}),\vec{t})$ . The trickiest thing is to keep track of the variables, in fact there is a *bug* in that incompatibilities between the sides is not handled: Example 5.1.3.F is not transformed.

```
1 xintroduction :: String → Rule → [Sym] → Rule
```

```
2 xintroduction z (lhs,rhs,nm) symbols =
```

```
3   (lhs, rhs', nm ++ (if rhs ≡ rhs' then "" else "-x")) where
```

```
4  rhs' = fst (xi [] [] [] rhs)
```

The work is done by the function that collects the metaapplications and leaves those alone that are applied to the same variable lists, replacing the rest with an explicit substitution variant. Each variable is identified by its metabstraction path to ensure uniqueness.

```
5  xi vps zs p t@(Mvar v) = (t,zs)
6  xi vps zs p (Mabs v t) =
7    (Mabs v t',zs') where (t',zs') = xi ((v,p+[0]):vps) zs (p+[0]) t
8  xi vps zs p (Mcon f ts) =
9    (Mcon f ts',zs') where (ts',zs') = xi' vps zs p 0 ts
10 xi vps zs p t@(Mapp m@(s,a,i) ts)
11   | mexplicit = (t,zst)
12   | otherwise = (Mcon (sigma a "")
13                 (foldr Mabs (Mapp m [Mvar v | v←vs]) vs : ts'),
14                 zs')
15   where (mexplicit,zst) | distvars ts [] = mx zs
16                       | otherwise      = (False,zs)
17   distvars :: [Mterm] → [Var] → Bool
18   distvars [] _ = True
19   distvars (Mvar v : ts) vs = v 'notElem' vs ∧ distvars ts (v:vs)
20   distvars _ _ = False
21   z = [ lookup vps v | Mvar v ← ts ]
22   mx :: [(Mv, [Path])] → (Bool, [(Mv, [Path])])
23   mx [] = (True, (m,z):zs)
24   mx ((m',z'):zs') | m ≡ m' ∧ z ≡ z' = (True,zs)
25                     | m ≡ m'       = (False,zs)
26                     | otherwise    = mx zs'
27   lookup [] _ = error"PANIC---can't find bound variable"
28   lookup ((v',p'):vps') v'' | v' ≡ v'' = p'
29                               | otherwise = lookup vps' v''
30   (ts',zs') = xi' vps zs p 0 ts
31   vs = take a (fresh [])
32   xi' vps zs p i [] = ([],zs)
33   xi' vps zs p i (t:ts) = (t':ts'',zs'')
34   where (t',zs') = xi vps zs (p+[i]) t
35         (ts'',zs'') = xi' vps zs' p (i+1) ts
36   sigma n s | fs 'elem' symbols = sigma n (s+"")
```

```

37         | otherwise           = fs
38         where fs = (z++s,n+1,False)

```

**6.7.3 Code (substitution distribution).** For each  $\Sigma^{n+1}$  symbol add a rule

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

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

$$\begin{aligned} \Sigma^{n+1}([\vec{x}]F^m(Z_1(\vec{x}), \dots, Z_m(\vec{x})), \vec{X}) & \quad (\text{x-F}^m\text{-n}) \\ \rightarrow F^m(\Sigma^{n+1}([\vec{x}]Z_1(\vec{x}), \vec{X}), \dots, \Sigma^{n+1}([\vec{x}]Z_m(\vec{x}), \vec{X})) \end{aligned}$$

```

1  xdistribute :: Sym → Sym → [Rule]
2  xdistribute f@(sf,af,_) s@(_,a,_) =
3    [ (Mcon s (foldr Mabs (Mcon f [ xapp n | n ← [1..af] ]) vs : mvapps),
4      Mcon f [ Mcon s (foldr Mabs (xapp n) vs : mvapps)
5              | n ← [1..af] ],
6      "x-$"++sf++"$-$"++show (a-1)+"$" ) ]
7  ++
8    [ (Mcon s (foldr Mabs (Mapp ("Z",a,False) vsyapps) vsy : mvapps),
9      Mabs "y" (Mcon s (foldr Mabs (Mapp ("Z",a,False) vsyapps) vs : mvapps)),
10     "xma-$"++show (a-1)+"$" ) ]
11  where vs = vectorx (a-1)
12        mvs = vectorX (a-1)
13        mvapps = [ Mapp (mv,0,False) [] | mv ← mvs ]
14        xapp n = if af ≡ 1 then Mapp ("Z",a,False) [ Mvar v | v ← vs ]
15                else Mapp ("Z_"++shownumb n,a,False) [ Mvar v | v ← vs ]
16        vsy = vs+["y"]
17        vsyapps = [Mvar v | v ← vsy]

```

**6.7.4 Code (substitution elimination).** For each  $\Sigma^{n+1}$  add for each  $i \in \{1, \dots, n\}$  two rules

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

where

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

(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).

```

1 xeliminate :: Sym → [Rule]
2 xeliminate s@(_,a,_) =
3   [ (Mcon s (foldr Mabs (Mvar (vs !! (j-1))) vs : mvapps),
4     Mapp (mvs !! (j-1),0,False) [],
5     "xv-$" ++ show i ++ "$-$" ++ show j ++ "$"
6   | j ← [1..a-1] ]
7   ++
8   [ (Mcon s (foldr Mabs (Mapp ("Z",i-1,False)
9     [ Mvar x | x ← vs' j ]) vs
10    : mvapps),
11    if i ≡ 1 then (Mapp ("Z",0,False) [])
12    else Mcon s (foldr Mabs (Mapp ("Z",i-2,False)
13      [ Mvar x | x ← vs' j ]) (vs' j)
14      : [ Mapp (mv,0,False) [] | mv ← mvs' j ]),
15    "xgc-$" ++ shownumb i ++ "$-$" ++ shownumb j ++ "$"
16  | j ← [1..a-1] ]
17   where i      = a-1
18         vs     = vectorx i
19         vs' j  = vectorx' i j
20         mvs    = vectorX i
21         mvs' j = vectorX' i j
22         mvapps = [ Mapp (mv,0,False) [] | mv ← mvs ]
23

```

### 6.7.5 Code (auxiliaries).

```

1 vectorx n      = if n ≡ 1 then ["x"] else [ "x_" ++ shownumb i | i ← [1..n] ]
2 vectorx' n n' = [ "x_" ++ shownumb i | i ← [1..n], i ≠ n' ]
3 vectorX n      = if n ≡ 1 then ["X"] else [ "X_" ++ shownumb i | i ← [1..n] ]
4 vectorX' n n' = [ "X_" ++ shownumb i | i ← [1..n], i ≠ n' ]

```

**6.7.6 Code (saturation).** Recall that a CRS is ‘saturated’ if all metaapplications in the rule patterns contain all the possible variables. This is coded directly as follows:

```

1 saturated :: CRS → Check
2 saturated crs = either "CRS_not_saturated:" (sat' crs) where
3   sat' [] = ""
4   sat' ((lhs,rhs,nm):rs) = sat1 [] lhs ++ sat' rs where
5     sat1 vs (Mvar v) = ""
6     sat1 vs (Mabs v t) = sat1 (v:vs) t
7     sat1 vs (Mcon _ ts) = concat [ sat1 vs t | t ← ts ]
8     sat1 vs t@(Mapp m ts)
9       | missingvars ≡ [] = ""
10      | otherwise = "\n'" ++ show t ++ "' should be '"
11                  ++ show(Mapp m (map Mvar vs))
12                  ++ "' in rule " ++ nm
13
14   where missingvars = vs \\ varsof ts
15         varsof [] = []
16         varsof (Mvar v:rest) = v : varsof rest
17         varsof _ = error"PANIC---variables_required_in_pattern"

```

## 6.8 Inferences

This section contains the implementation of a *CRS inference rule* procedure.<sup>12</sup>

Our solution is inspired by *Petri nets* (Danthine 1980) in that we insert in an otherwise contextual system special ‘tokens’ into terms that represent the ‘active point’ of evaluation; a similar idea is used in the generalised graph rewriting language DACTL language of Glauert, Kennaway and Sleep (1989) where ‘active markers’ are required for evaluation to proceed. We accept rules in the simple form of  $LHS \rightarrow RHS$  where the premises are *inserted at the location of their LHS in the LHS of the entire rule* – then the scoping fits.

**6.8.1 Code (inference rules).** This procedure transforms a pseudo-CRS which includes ‘inference rules’ of the above form.

```

1 inferencify :: CRS → CRS

```

The main function merely generates the generic ‘unload’ rule and iterates over all CRS’s rules.

```

2 inferencify crs = assert (checkCRS crs') crs' where
3   crs' = cleanup (inf 0 crs) ++ [(root (overline mZ), clean mZ, "done")]

```

<sup>12</sup>The section is the result of typesetting the literate Haskell script “crsi.lhs”.

```

4       where mZ= Mapp ("Z",2,False) []
5     inf _ []      = []
6     inf n (r:rs) = inf1 n1 r++inf n1 rs where n1=n+1

```

For each rule we distinguish between axioms and inferences: axioms are distinguished by having no occurrences of the nested  $\rightarrow$  symbol.

```

7     inf1 n (lhs,rhs,nm)
8       | premises  $\equiv$  [] = [startrule,
9                             (underline lhs, overline rhs, nm)]
10      | otherwise = [startrule,
11                      (underline (dist head lhs),
12                        dist (underline.head) lhs, nmadd nm"1"),
13                      (dist (overline.last) lhs,
14                        overline rhs, nmadd nm"2")]
15     where

```

Every rule adds its LHS to the patterns that denote reducible terms.

```

16     startrule =
17       (clean (dist head lhs), root (underline (dist head lhs)),nmadd nm"0")

```

The premises nested in a LHS are the submetaterms using a  $\rightarrow$ -construction except we have to reduce the variable arguments to only those actually part of the premise.

```

18     premises = enum' 1 (prem [] lhs) where
19       prem vs (Mabs v t) = prem (v:vs) t
20       prem vs (Mcon ("to",2,True) [pat,cont]) =
21         [(prem' vs pat, prem' vs cont, nmadd nm "\\dn")]
22       prem vs (Mcon _ ts) = concat [ prem vs t | t  $\leftarrow$  ts ]
23       prem vs _ = []
24       prem' vs (Mvar v)      = Mvar v
25       prem' vs (Mabs v t)    = Mabs v (prem' (vs'but' [v]) t)
26       prem' vs (Mcon f ts)  = Mcon f [ prem' vs t | t  $\leftarrow$  ts ]
27       prem' vs (Mapp (s,a,i) ts) =
28         Mapp (s,length ts',i) ts' where
29           ts' = removevs ts
30           removevs [] = []
31           removevs (Mvar v : xs) | v'elem'vs = removevs xs
32                                   | otherwise = Mvar v : removevs xs
33           removevs _ = error"non-variable_pattern_metaapplication_argument"

```

In order to obtain unique names we have to add sequence numbers to the premises.

```

34     enum' n [] = []
35     enum' n ((lhs',rhs',nm'):rs') =
36         (lhs',rhs',nmadd nm' (show n)) : enum' (n+1) rs'

```

Last the elimination rule for the inference rule is just the inference where the premises have been replaced with their overlined results.

```

37     dist mk (Mvar v) = Mvar v
38     dist mk (Mabs v t) = Mabs v (dist mk t)
39     dist mk (Mcon ('{to}',2,True) premise) = mk premise
40     dist mk (Mcon f ts) = Mcon f [ dist mk t | t ← ts ]
41     dist mk (Mapp m ts) = Mapp m ts

```

Here are the auxiliaries.

```

42     clean      t = Mcon markerfs [root t]
43     root      t = Mcon markerfs [Mcon markerfs [t]]
44     underline t = Mcon underlinefs [Mcon markerfs [t]]
45     overline  t = Mcon overlinefs [Mcon markerfs [t]]

46     underlinefs = ("\\u",1,False)
47     overlinefs  = ("\\h",1,False)
48     markerfs    = ("'<>",1,False)

49     nmadd nm s | last nm ≡ '$' = init nm ++ s ++ "$"
50               | otherwise     = nm ++ "$" ++ s ++ "$"

```

Cleaning up is merely removing trivial things.

```

51     cleanup [] = []
52     cleanup (r@(lhs',rhs',nm):rs)
53         | lhs' ≡ rhs' = cleanup rs
54         | otherwise = r : filter (λ(lhs',rhs',_) → lhs' ≠ lhs' ∨ rhs' ≠ rhs')
55                               (cleanup rs)

```

## 6.9 Main Program & Bootstrap

In this section we present the main program module<sup>13</sup> with the functions making it possible to actually run the CRS implementation presented in this chapter on a computer using the HUGS Haskell system of Jones (1995).

Most of the definitions in this section are concerned with *interactive* CRS reduction. Last in the section we present the fragments that bind the CRS reducer together.

---

<sup>13</sup>The section is the result of typesetting the literate Haskell script “crsmain.lhs”.

**6.9.1 Code (documentation).** Message for the user of the test system (see Figure A.1, page 209, for the resulting output).

```

1 welcome = unlines
2   ["",
3     "Welcome to the CRS interpreter (version " ++ version ++ ")",
4     "Copyright (c) 1995-1996 Kristoffer H. Rose <kris@diku.dk>."]
5 rcs = words "$Id: crsmain.lhs,v 2.13 1996/02/07 19:48:49 kris Exp kris $"
6 version = rcs !! 2 ++ " of " ++ rcs !! 3
7         ++ if length rcs > 7 then " locked by " ++ (rcs !! 7) else ""
8 help = welcome ++ unlines
9   ["",
10    "ACTIVATION:",
11    "?_load_<crs>_make_<crs>_the_current_crs1",
12    "?_see_<crs>_show_<crs>_on_screen",
13    "?_save_<crs>_save_current_crs1_as_<crs>_(must_be_single_file_-)",
14    "",
15    "<crs>_should_be_a_string_with_names_of_files_that_contain_CRSs",
16    "separated_by_blanks,_<num>_for_the_current_crs<num>,_a",
17    "transformation,_or_\"CRS:...\\"_where_..._is_an_entire_CRS.",
18    "",
19    "RUNNING:",
20    "?_reds_<term>_show_redexes_in_<term>",
21    "?_step_<term>_one_step_parallel_outermost_reduction_of_<term>",
22    "?_try_<term>_parallel_outermost_reduction_of_<term>_to_nf",
23    "?_tryi_<term>_parallel_innermost_reduction_of_<term>_to_nf",
24    "?_search_<term>_exhaustive_single_reductions_of_<term>_to_nf",
25    "",
26    "<term>_should_be_a_CRS_metaterm;_the_current_crs1_CRS_is_used.",
27    "",
28    "These_exist_in_variants_where_<strategy>_and/or_<crs>_can_be",
29    "explicitly_specified_(variants_with_CS_requiring_both_exist):",
30    "?_stepC_<crs>_<term>_?_steps_<strategy>_<term>",
31    "?_tryC_<crs>_<term>_?_trys_<strategy>_<term>",
32    "?_searchC_<crs>_<term>_?_searchS_<strategy>_<term>",
33    "",
34    "<strategy>_should_be_one_of_'innermost',_'outermost',_'weak',",
35    "'strong',_'leftmost',_or_'rightmost'.",
36    "",
37    "In_these_commands_C_can_be_replaced_by_C2_and_then_two_<crs>_args",
38    "are_needed:_the_second_is_applied_completely_between_reductions.",
39    "",
40    "CHECKS:",
41    "?_laps_<crs>_show_overlaps_between_<crs>_rule_patterns",

```



```

42  "?_crits_<crs>_show_critical_pairs_of_<crs>_rules",
43  "?_orth_<crs>_check_that_<crs>_is_orthogonal",
44  "?_worth_<crs>_check_that_<crs>_is_weakly_orthogonal",
45  "?_lc_<crs>_approximate_search_for_proof_of_<crs>_LC",
46  "?_sat_<crs>_check_<crs>_saturated",
47  "",
48  "TRANSFORMATIONS_(yield_new_<crs>s):",
49  "?_expl_<crs>_<crs>_after_explicification",
50  "?_infer_<crs>_<crs>_after_internalising_inference_rules",
51  "?_conv_<crs>_<crs>_after_reversing_all_rules",
52  "?_mini_<crs>_<crs>_after_removing_redundant_rules"]

```

**6.9.2 Code (interactive operation).** The implementation of the documented functions can be used directly from the HUGS command line. This keeps a *current working CRS* in a file which is loaded and viewed using these operations.

```

1  load c = loadCRS2 (crs c) []
2  load2 c c2 = loadCRS2 (crs c) (crs c2)
3  loadCRS crs = writeFile "crs1.crs" (showCRS crs) complain done
4  loadCRS2 crs crs2 = writeFile ("crs1.crs") (showCRS crs) complain $
5                      writeFile ("crs2.crs") (showCRS crs2) complain done
6  see c = showCRS (crs c)

```

**6.9.3 Code (CRS input).** The actual reading of a CRS from the command line or a file is handled by these functions. The central one is `crs` which understands the `<crs>` format described in the help text: either the current working CRS,

```

1  crsname s | s ≡ "" = "crs1.crs"
2            | all ('elem' ['0'..'9']) s = "crs"+s+".crs"
3            | otherwise = s+".crs"

```

The `read` function has one more possibility, namely a CRS directly on the command line.

```

4  crs "" = crsf ""
5  crs s@( 'C': 'R': 'S': ':': ':': '_' ) = crs' s
6  crs s@( '%' : 'C': 'R': 'S': ':': ':': '_' ) = crs' s

```

or the CRS obtained by concatenating the CRSs of several files.

```

7  crs fs = case map crsf (words fs) of
8          [c] → c
9          cs → assert(checkCRS c) c where c = concat cs

```

The simple input is handled by these functions.

```

10 crss s = crs' ("CRS:␣"++s)
11 crsf fn = crs' (openfile (crsname fn))
12 crs' s = case readParseTree s of
13         Crs c → c
14         _     → error "INPUT_MISTAKE---not_a_<crs>"

```

**6.9.4 Code (metaterm input).** Similarly, metaterms can be typed on the command line or in a file, however, there is no automatic handling of this.

```

1 term s = term' ("METATERM:␣"++s)
2 termf fn = term' (openfile (fn+".term"))
3 term' s = case readParseTree s of
4         MetaTerm t → t
5         _          → error "INPUT_MISTAKE---not_a_<term>"

```

**6.9.5 Code (CRS output).** We can also write CRSs.

```

1 save "" = done
2 save fn = writeFile (crsname fn) (see"") complain done

```

**6.9.6 Code (reduction).** These experiment with reduction of terms using the current working CRS

Next the execution commands; the versions without show

Reducing with some strategy a single step or until normal form means just that. Only good for deterministic nonoverlapping strategies.

```

1 reds s = show (redexes (crsf "crs1") (term s))
2 step s = show (onestep (crsf "crs1") (crsf "crs2") outermost (term s))
3 try s = show (nf (crsf "crs1") (crsf "crs2") outermost (term s))
4 tryi s = show (nf (crsf "crs1") (crsf "crs2") innermost (term s))
5 search s =
6   unlinesbeep (map show (nfs (crsf "crs1") (crsf "crs2") strong (term s)))
7 stepS strat s = show (onestep (crsf "crs1") (crsf "crs2") strat (term s))
8 tryS strat s = show (nf (crsf "crs1") (crsf "crs2") strat (term s))
9 searchS strat s =
10  unlinesbeep (map show (nfs (crsf "crs1") (crsf "crs2") strat (term s)))
11 redsC c s = show (redexes (crsf c) (term s))
12 stepC c s = show (onestep (crs c) [] outermost (term s))

```

```

13 tryC c s = show (nf (crs c) [] outermost (term s))
14 tryiC c s = show (nf (crsf c) [] innermost (term s))
15 searchC c s =
16   unlinesbeep (map show (nfs (crs c) (crsf "crs2") strong (term s)))

17 redsC2 c c2 s = show (redexes (crsf c) (term s))
18 stepC2 c c2 s = show (onestep (crs c) (crs c2) outermost (term s))
19 tryC2 c c2 s = show (nf (crs c) (crs c2) outermost (term s))
20 tryiC2 c c2 s = show (nf (crsf c) (crs c2) innermost (term s))
21 searchC2 c c2 s =
22   unlinesbeep (map show (nfs (crs c) (crs c2) strong (term s)))

23 stepC2S c c2 strat s = show (onestep (crs c) (crs c2) strat (term s))
24 tryC2S c c2 strat s = show (nf (crs c) (crs c2) strat (term s))
25 searchC2S c c2 strat s =
26   unlinesbeep (map show (nfs (crs c) (crs c2) strat (term s)))

27 unlinesbeep = concat . map (++ "\a\n")

```

**6.9.7 Code (analysis).** These read CRSs and terms from strings and files.

```

1 laps c = showOverlaps (overlaps (crs c))
2 crits c = unlines (map showcrits (criticalpairs (crs c))) where
3   showcrits (l,r,i) = "\nrules_" ++ i
4                       ++ "\ngenerate_" ++ show l ++ "\n<->" ++ show r
5 orth c = assert (orthogonal (crs c)) "CRS_is_orthogonal."
6 worth c = assert (weaklyorthogonal (crs c)) "CRS_is_weakly_orthogonal."
7 sat c = assert (saturated (crs c)) "CRS_is_saturated."
8 lc c = assert (locallyconfluent (crs c)) "CRS_is_locally_confluent."

```

**6.9.8 Code (transformations).**

```

1 expl c = showCRS (explicitify (crs c))
2 infer c = showCRS (inferencify (crs c))
3 conv c = showCRS (converse (crs c))
4 mini c = showCRS (minimalise (crs c))

```

**6.9.9 Code (bootstrap).** In order to run the system with HUGS the file `crs.prj` contains a list of all the program files:

```

-- $Id: crs.prj,v 2.4 1996/01/29 19:51:45 kris Exp kris $--hugs--

-- CRS interpreter.
-- Copyright © 1995-1996 Kristoffer Høgsbro Rose, all rights reserved.

hugs.hs          -- HUGS-specific primitives
idioms.lhs       -- idioms
crstype.lhs      -- datatypes
crsinput.lhs     -- input
crs_scan.hs      -- scanner generated from crs.gram by Ratatosk
crs_parse.hs     -- parser generated from crs.tokens by Ratatosk
crsred.lhs       -- reduction
crsstrat.lhs    -- strategies
crsanal.lhs     -- analysis
crsx.lhs        -- explicification
crsi.lhs        -- inference rules
crsmain.lhs     -- main program & bootstrap

crsprettty.lhs  -- prettyprinting of term with redexes (for thesis)

```

Two of these have not been shown above. The first, `hugs.hs`, contains declarations of a few missing primitives as follows.

```

-- hugs.hs: HUGS-specific primitives!

primitive primFopen "primFopen" :: String → a → (String → a) → a
openfile :: String → String
openfile f = primFopen f (error ("can't open file" ++ f)) id

primitive trace "primTrace" :: String → a → a

```

The second, `crsprettty.lhs` contains a variant of the `showMterm` function of Code 6.2.10 that prints terms with redexes such as shown in Example 2.6.16; it is reproduced in the appendix A.4.

## 6.10 Summary

The program that we have presented works as has been demonstrated, but the CRS system has not been subject to extensive testing. Yet we have found that working in a high-level language such as Haskell has significantly eased the task of programming based on the formula of this dissertation, in particular the rapid prototyping that is possible in Haskell, first of all in the literate Haskell programming style, has been a good advantage.

# 7

## Conclusions

We have achieved a number of milestones with this thesis. In general, we have described a consistent and comprehensive set of theoretical tools for reasoning operationally about reduction that can even be combined in different ways such as to obtain different levels of abstraction on operational aspects of reduction of functional programs.

Specifically, a major milestone is the invention and analysis of the  $\lambda x(gc)$  family of calculi: with this we have achieved a pleasant coherence in descriptions of a range of issues, both reproducing known results such as the abstract machines, new results, such as the direct proof of  $\lambda x \text{ PSN } \lambda \beta$ , and of course the classification which in hindsight is what made it possible.

Furthermore, an issue which we believe deserve impact on the theoretical computer science community at large, is the current rediscovery of *higher-order rewriting* by several groups; in this work we have contributed to this by relating the CRS formalism to operational semantics.

The detailed enumeration of our contributions is located in section 1.1.

**Personal Perspective.** This work has grown significantly beyond the original scope, which was “merely” to investigate the operational theory of sharing. We have, instead, partly succeeded in a much more thorough study of the *fundamental interaction between operational aspects of functional programming languages*. The major reason for the large scope of the present dissertation is that

the author (re)discovered the CRS formalism and found that a very large number of systems, derivors, translators, *etc.*, could be expressed as CRS and then – executed!

This sidetrack led to the main insight of this thesis (in the author’s opinion) that the operational aspects of reduction and evaluation are largely orthogonal and can be combined at leisure – so it is not a question of “whether de Bruijn indices are better/more fundamental/cleaner/*etc.* for property  $X$ ” but rather a question of “which dimension is the critical one for  $X$ ”. This is new territory, and the search for interesting points in explicitness space has been sometimes frustrating, sometimes rewarding!

**Future directions.** The problem when “opening a space”, as this work has attempted, is that a space has many points, even when some of the dimensions are discrete! And this is indeed the case for this work. There are several major directions for future work that the author would like to continue studying.

- A. The classification of *naming paradigms* opens up a large number of possibilities for relating different storage architectures, of which the simplest are stacks and heaps, of course, however, much more could be done.
- B. It seems feasible to generalise the notion of reduction strategy to CRSs in an internal way (in fact our preliminary studies in this direction are implicitly included in section 6.8). It may even be possible to derive machines and other flat reduction systems automatically by automating the machine derivation principle we have outlined.
- C. The issue of *complexity* definitively deserves further study beyond the mere hints that sharing
- D. Finally, it seems that a higher-level description of pattern matching than is usual can facilitate a much smoother methodology for program transformations by exploiting referential transparency of pure functional languages to a larger extent.

Also we have not at all studied type systems or other static aspects of functional programming languages.

**One** thing is certain, though: rewriting is interesting, rewarding, and much more versatile than the author had imagined.



```

8 Reading script file "/usr/local/lib/hugs/hugs.prelude":
9 Hugs session for:
10 /usr/local/lib/hugs/hugs.prelude
11 Type :? for help

```

Now HUGS is started it prompts us with a ? and we can load the CRS ‘project file’ shown in Code 6.9.9.

```

? :project crs.prj

12 Reading script file "hugs.hs":
13 Reading script file "idioms.lhs":
14 Reading script file "crstype.lhs":
15 Reading script file "crsinput.lhs":
16 Reading script file "crs_scan.hs":
17 Reading script file "crs_parse.hs":
18 Reading script file "crsred.lhs":
19 Reading script file "crsstrat.lhs":
20 Reading script file "crsanal.lhs":
21 Reading script file "crsx.lhs":
22 Reading script file "crsi.lhs":
23 Reading script file "crsmain.lhs":
24 Reading script file "crsprettty.lhs":

25 Type :? for help

```

At this point we can ask the interpreter for assistance; the result is shown in Figure A.1.

Of all these commands we will first make the  $\lambda$ -calculus of Example 2.6.6 the default CRS, extended with the  $\lambda$ -terms corresponding to the standard combinators of Notation 2.3.8. They reside in the files `lambda.crs` and `ski.crs` (that were in fact used to typeset the mentioned definitions), so we should concatenate the two files into the ‘current’ CRS.

```
? load"lambda ski"
```

HUGS prints the cost of every operation in this form.

```
64 (24405 reductions, 74076 cells)
```

We can check that we have, indeed, loaded the right CRS as follows, exploiting that the ‘current’ CRS is denoted "":

```
? see""
```

```
65 CRS:
```

```
66 ('lx.Z(x))Y &"->" Z(Y) \tag{${'b$}}\
```



```

? help

26 Welcome to the CRS interpreter (version 3.0 of 1996/02/09)!
27 Copyright © 1995-1996 Kristoffer H. Rose <kris@diku.dk>.

28 ACTIVATION:
29 ? load <crs>          make <crs> the current #1
30 ? see <crs>          show <crs> on screen
31 ? save <crs>         save current #1 as <crs> (must be single file :-))

32 <crs> should be a string with names of files that contain CRSs
33 separated by blanks, <num> for the current #<num>, a
34 transformation or "CRS: ..." where ... is an entire CRS.

35 RUNNING:
36 ? reds <term>        show redexes in <term>
37 ? step <term>       one step parallel outermost reduction of <term>
38 ? try <term>        parallel outermost reduction of <term> to nf
39 ? tryi <term>       parallel innermost reduction of <term> to nf
40 ? search <term>     exhaustive single reductions of <term> to nf

41 <term> should be a CRS metaterm; the current #1 CRS is used.

42 These exist in variants where <strategy> and/or <crs> can be
43 explicitly specified (variants with CS requiring both exist):
44 ? stepC <crs> <term>    ? stepS <strategy> <term>
45 ? tryC <crs> <term>    ? tryS <strategy> <term>
46 ? searchC <crs> <term>  ? searchS <strategy> <term>

47 <strategy> should be one of 'innermost', 'outermost', 'weak',
48 'strong', 'leftmost', or 'rightmost'.

49 In these commands C can be replaced by C2 and then two <crs> args
50 are needed: the second is applied completely between reductions.

51 CHECKS:
52 ? laps <crs>          show overlaps between <crs> rule patterns
53 ? crits <crs>        show critical pairs of <crs> rules
54 ? orth <crs>         check that <crs> is orthogonal
55 ? worth <crs>        check that <crs> is weakly orthogonal
56 ? lc <crs>           approximate search for proof of <crs> LC
57 ? sat <crs>          check <crs> saturated

58 TRANSFORMATIONS (yield new <crs>s):
59 ? expl <crs>         <crs> after explicification
60 ? infer <crs>        <crs> after internalising inference rules
61 ? conv <crs>         <crs> after reversing all rules
62 ? mini <crs>         <crs> after removing redundant rules

63 (5287 reductions, 16628 cells)

```

Figure A.1: Help message from CRS interpreter.

```

67 I &"->" 'lx.x \tag{I$}\
68 K &"->" 'lxy.x \tag{K$}\
69 {K^*} &"->" 'lxy.y \tag{{K^*}$}\
70 S &"->" 'lxyz.xz(yz) \tag{S$}\
71 'W &"->" 'w'w \tag{'W$}\
72 'w &"->" 'lx.xx \tag{'w$}\
73 {Y} &"->" 'lf.('lx.f(xx))('lx.f(xx)) \tag{{Y}$}\
74 'Q &"->" AA \tag{'Q$}\
75 A &"->" 'lxy.y(xxy) \tag{A$}

76 (17719 reductions, 51066 cells)

```

It should be apparent to the  $\text{\LaTeX}$  (Lamport 1994) user that  $\mathcal{A}\mathcal{M}\mathcal{S}\text{-}\text{\LaTeX}$  (American Mathematical Society 1995) equation system format is used for this in that `\` separates rules and `\tag` is used to name them.<sup>1</sup> More importantly, however, it reveals that the interpreter uses the syntax `'l` for  $\lambda$  and `"->"` for the reduction arrow, conventions of the `qsymbols`  $\text{\LaTeX}$  package (Rose 1994).

Now we can check that the system is weakly orthogonal so that it is safe to use the standard reduction strategy, namely parallel outermost.

```
? worth"
```

```

77 CRS is weakly orthogonal.
78 (17694 reductions, 48832 cells)

```

Now let us 'try' a simple reduction, using the default strategy

```
? try"SKK"
```

This first produces a trace output (which may be disabled) describing each reduction step with the step number and number of redexes, and then all the redexes on the form `{path:rule}`.

```

79 %% 1/3 {00:S$} {01:K$} {1:K$}
80 %% 2/1 {0:'b$}
81 %% 3/1 {:'b$}
82 %% 4/2 {00:'b$} {001:'b$}
83 %% 5/1 {00:'b$}

```

Finally the 'real' output: the normal form.

```

84 'la.a
85 (27809 reductions, 77987 cells)

```

---

<sup>1</sup>This is how we have been able to claim throughout this dissertation that systems showed are really those used for reduction.

In order to find out what this looks like as a combinator we can try to find SKI combinator combinations that create this – this is done with the following command (see the help text above for an explanation).

```
? tryC (conv "ski") "'la.a"
```

As expected the result is I after one reduction (which is an ‘expansion’ in the "ski" system).

```
86 %% 1/1 {:$I$}
87 I
88 (34648 reductions, 100470 cells)
```

That’s all for this example.

```
? :quit
89 [Leaving Hugs]
```

**A.1.2 Example (prettyprinting).** The pretty-printing reduction of A.4 is used as follows to reproduce the  $\beta$ -reduction of Example 3.1.8. Those commands that exist in a ‘pretty-printing’ version have a pp prefix. What we want to do is a complete search of all single rewrites of the  $\lambda$ -term  $(\lambda x.(\lambda y.y)x)x$ . This, however, has a free variable so we need to mark the last  $x$  as a *constructor* by surrounding it with  $\{ \}$ s. Here is how:

```
? load"lambda"
1 (27381 reductions, 82160 cells)
? pptryS strong "('lx.('ly.y)x){x}"
2 \Metaterm{
3 \markredex{\('lx. \markredex{\('ly.y)x}{'$b$} \){x}}{'$b$}
4 }%
5 \Reduction{"->!P"}%% 1/2 {:$'b$} {000:$'b$}
6 \Metaterm{
7 \markredex{\('la.a){x}}{'$b$}
8 }%
9 \Reduction{"->!"}%% 2/1 {:$'b$}
10 \Metaterm{
11 {x}
12 }%
13 (9550 reductions, 22047 cells)
```

As should be clear, ‘pretty-printing’ here means that it does not become readable until run through  $\text{\LaTeX}$  in a suitable environment; the result is shown below.

$$\boxed{\lambda x. \boxed{\lambda y. y} x} \xrightarrow{(\beta)} \boxed{\lambda a. a} x \longrightarrow x$$

(β) (β)

**A.1.3 Code (Fibonacci number).** The following session establishes the system of Definition 1.3.1 as a CRS on Peano numbers. First we enter (on a single line) the hugs command

```
? load "CRS:
?   {fib}(0) → 0 TAG{F0} ; {fib}(S(0)) → S(0) TAG{F1} ;
?   {fib}(S(S(X))) → {fib}(S(X))+{fib}(X) TAG{Fn} ;
?   X+0 → X TAG{+0} ; X+S(Y) → S(X)+Y TAG{+1}"
1 (9513 reductions, 27318 cells)
```

We save this system for later use.

```
2 save"fib+"
? (11088 reductions, 32126 cells)
```

In the CRS file `1to30.crs` we have the system

```
1 → S(0); 2 → S(1); 3 → S(2); 4 → S(3); 5 → S(4);
6 → S(5); 7 → S(6); 8 → S(7); 9 → S(8); 10 → S(9);
11 → S(10); 12 → S(11); 13 → S(12); 14 → S(13); 15 → S(14);
16 → S(15); 17 → S(16); 18 → S(17); 19 → S(18); 20 → S(19);
21 → S(20); 22 → S(21); 23 → S(22); 24 → S(23); 25 → S(24);
26 → S(25); 27 → S(26); 28 → S(27); 29 → S(28); 30 → S(29)
```

With these we can now find numbers in the fibonacci sequence and even see them in a readable notation. Here is the computation of the sixth number.

```
? tryC(conv"1to30") (tryC"fib+_1to30" "{fib}(6)")
...
3 8
4 (401064 reductions, 1221596 cells, 1 garbage collection)
```

## A.2 λ-calculus

This demonstrates the result of running the CRS system of chapter 6 on the  $\lambda_{xgc}$ -calculus of chapter 3. The system is shown in Definition 3.1.4, however, we show the CRS as it is used here below.

### A.2.1 Example.

The  $\lambda$ xgc-reduction CRS is split in a file for each subrelation.

```
? load "b x gc"
```

```
1 (27381 reductions, 82160 cells)
```

The system looks like this to the CRS interpreter:

```
? see"
```

```
2 CRS:
```

```
3 ('lx.\mS{M}(x))\m{N} &"->" \XS ([x]\mS{M}(x),(\m{N})~\fresh ) \tag{ba}\
4 \XS ([x]x,\m{N}) &"->" \m{N} \tag{xv}\
5 \XS ([x]\mV{y},\m{N}) &"->" \mV{y} \tag{xvgc}\
6 \XS ([x]'ly.\mS{M}(x,y),\m{N}) &"->" 'ly.\XS ([x]\mS{M}(x,y),\m{N}) \tag{xab}\
7 \XS ([x](\mS{M}_1(x))(\mS{M}_2(x)),\m{N})
8 &"->" \XS ([x]\mS{M}_1(x),\m{N})\XS ([x]\mS{M}_2(x),\m{N}) \tag{xap}\
9 (\m{M})~\m{a}\m{N} &"->" \m{M}\m{N} \tag{cpap}\
10 \XS ([x](\mS{M}(x))~\m{a},\m{N}) &"->" \XS ([x]\mS{M}(x),\m{N}) \tag{cpx}
```

```
11 (81886 reductions, 164620 cells)
```

As can be seen, the substitution  $(M)\langle x := N \rangle$  is written  $\backslash XS \{[x]M,N\}$ , thus in almost genuine CRS notation which is then converted by  $\text{T}_{\text{E}}\text{X}$  to the usual notation. Here is the reductions of Figure 3.2 for this system.

```
? ppsearch"('lx.('ly.y)x){x}"
```

... *rewrite output typeset below* ...

```
12 [{x}]
```

```
13 (340781 reductions, 622543 cells)
```

$$\boxed{(\lambda x.(\lambda y.y)x)x} \longrightarrow ((\lambda b.b)a)\langle a := x \rangle \quad (\lambda x.(\lambda y.y)x)x \longrightarrow (\lambda x.(a)\langle a := x \rangle)x$$

(b) (b)

$$\boxed{((\lambda b.b)a)\langle a := x \rangle} \longrightarrow (\lambda b.b)\langle a := x \rangle(a)\langle a := x \rangle$$

(xap)

$$\boxed{((\lambda b.b)a)\langle a := x \rangle} \longrightarrow ((b)\langle b := a \rangle)\langle a := x \rangle \quad \boxed{(\lambda x.(a)\langle a := x \rangle)x} \longrightarrow ((b)\langle b := a \rangle)\langle a := x \rangle$$

(b) (b)

$$(\lambda x.(\lambda y.y)x)x \longrightarrow (\lambda x.x)x \quad \boxed{(\lambda b.b)\langle a := x \rangle}(a)\langle a := x \rangle \longrightarrow (\lambda a.(a)\langle b := x \rangle)(a)\langle a := x \rangle$$

(xv) (xab)

$$\boxed{(\lambda b.b)\langle a := x \rangle}(a)\langle a := x \rangle \longrightarrow (\lambda a.a)(a)\langle a := x \rangle$$

(gc)

$$(\lambda b.b)\langle a := x \rangle \boxed{(a)\langle a := x \rangle} \longrightarrow ((\lambda b.b)\langle a := x \rangle)x \quad \boxed{(b)\langle b := a \rangle}\langle a := x \rangle \longrightarrow (a)\langle a := x \rangle$$

(xv) (xv)

$$\boxed{(\lambda x.x)x} \longrightarrow (a)\langle a := x \rangle \quad \boxed{(\lambda a.(a)\langle b := x \rangle)(a)\langle a := x \rangle} \longrightarrow ((a)\langle b := x \rangle)\langle a := (a)\langle a := x \rangle \rangle$$

(b) (b)

$$(\lambda a.(\lambda y.y)x)\langle a := x \rangle \longrightarrow (\lambda a.a)(a)\langle a := x \rangle$$

(xvgc)

$$(\lambda a.(\lambda y.y)x)\langle a := x \rangle \longrightarrow (\lambda a.a)(a)\langle a := x \rangle$$

(gc)

$$\begin{aligned}
& (\lambda a.(a)\langle b := x \rangle) \boxed{(a)\langle a := x \rangle} \longrightarrow (\lambda a.(a)\langle b := x \rangle)x \\
& \hspace{10em} \text{(xv)} \\
& \boxed{(\lambda a.a)(a)\langle a := x \rangle} \longrightarrow (a)\langle a := (a)\langle a := x \rangle \rangle \quad (\lambda a.a) \boxed{(a)\langle a := x \rangle} \longrightarrow (\lambda a.a)x \\
& \hspace{10em} \text{(b)} \hspace{10em} \text{(xv)} \\
& \boxed{((\lambda b.b)\langle a := x \rangle)}x \longrightarrow (\lambda a.(a)\langle b := x \rangle)x \quad \boxed{((\lambda b.b)\langle a := x \rangle)}x \longrightarrow (\lambda a.a)x \\
& \hspace{10em} \text{(xab)} \hspace{10em} \text{(gc)} \\
& \boxed{(a)\langle a := x \rangle} \longrightarrow x \quad \boxed{((a)\langle b := x \rangle)}\langle a := (a)\langle a := x \rangle \rangle \longrightarrow (a)\langle a := (a)\langle a := x \rangle \rangle \\
& \hspace{10em} \text{(xv)} \hspace{10em} \text{(xvgc)} \\
& \boxed{((a)\langle b := x \rangle)}\langle a := (a)\langle a := x \rangle \rangle \longrightarrow (a)\langle a := (a)\langle a := x \rangle \rangle \\
& \hspace{10em} \text{(gc)} \\
& ((a)\langle b := x \rangle)\langle a := \boxed{(a)\langle a := x \rangle} \rangle \longrightarrow ((a)\langle b := x \rangle)\langle a := x \rangle \\
& \hspace{10em} \text{(xv)} \\
& \boxed{(\lambda a.a)(a)\langle a := x \rangle} \longrightarrow (a)\langle a := (a)\langle a := x \rangle \rangle \quad (\lambda a.a) \boxed{(a)\langle a := x \rangle} \longrightarrow (\lambda a.a)x \\
& \hspace{10em} \text{(b)} \hspace{10em} \text{(xv)} \\
& \boxed{(\lambda a.(a)\langle b := x \rangle)}x \longrightarrow ((a)\langle b := x \rangle)\langle a := x \rangle \quad (\lambda a.\boxed{(a)\langle b := x \rangle})x \longrightarrow (\lambda a.a)x \\
& \hspace{10em} \text{(b)} \hspace{10em} \text{(xvgc)} \\
& (\lambda a.\boxed{(a)\langle b := x \rangle})x \longrightarrow (\lambda a.a)x \quad \boxed{(a)\langle a := (a)\langle a := x \rangle \rangle} \longrightarrow (a)\langle a := x \rangle \\
& \hspace{10em} \text{(gc)} \hspace{10em} \text{(xv)} \\
& (a)\langle a := \boxed{(a)\langle a := x \rangle} \rangle \longrightarrow (a)\langle a := x \rangle \quad \boxed{(\lambda a.a)x} \longrightarrow (a)\langle a := x \rangle \\
& \hspace{10em} \text{(xv)} \hspace{10em} \text{(b)} \\
& \boxed{(a)\langle a := (a)\langle a := x \rangle \rangle} \longrightarrow (a)\langle a := x \rangle \quad (a)\langle a := \boxed{(a)\langle a := x \rangle} \rangle \longrightarrow (a)\langle a := x \rangle \\
& \hspace{10em} \text{(xv)} \hspace{10em} \text{(xv)} \\
& \boxed{((a)\langle b := x \rangle)}\langle a := x \rangle \longrightarrow (a)\langle a := x \rangle \quad \boxed{((a)\langle b := x \rangle)}\langle a := x \rangle \longrightarrow (a)\langle a := x \rangle \\
& \hspace{10em} \text{(xvgc)} \hspace{10em} \text{(gc)} \\
& \boxed{(\lambda a.a)x} \longrightarrow (a)\langle a := x \rangle \quad \boxed{(a)\langle a := x \rangle} \longrightarrow x \quad \boxed{(a)\langle a := x \rangle} \longrightarrow x \\
& \hspace{10em} \text{(b)} \hspace{10em} \text{(xv)} \hspace{10em} \text{(xv)}
\end{aligned}$$

### A.2.2 Example.

We can also run the example of Figure 3.4 using the modulo feature of the CRS system:

```

? load2 "b x" "gc"
1 (74445 reductions, 160661 cells)
? ppsearch("(lx.(ly.y)x){x}"
... rewrite output typeset below ...
2 [{x}]
3 (126321 reductions, 248958 cells)

```

Here are the reductions.

$$\begin{array}{l}
\boxed{(\lambda x. (\lambda y. y) x) x} \xrightarrow{(b)} ((\lambda b. b) a) \langle a := x \rangle \quad (\lambda x. \boxed{(\lambda y. y) x}) x \xrightarrow{(b)} (\lambda x. (a) \langle a := x \rangle) x \\
\boxed{((\lambda b. b) a) \langle a := x \rangle} \xrightarrow{(xap)} (\lambda b. b) \langle a := x \rangle (a) \langle a := x \rangle \\
\boxed{((\lambda b. b) a) \langle a := x \rangle} \xrightarrow{(b)} ((b) \langle b := a \rangle) \langle a := x \rangle \quad \boxed{(\lambda x. (a) \langle a := x \rangle) x} \xrightarrow{(b)} ((b) \langle b := a \rangle) \langle a := x \rangle \\
(\lambda x. \boxed{(a) \langle a := x \rangle}) x \xrightarrow{(xv)} (\lambda x. x) x \quad \boxed{(\lambda a. a) (a) \langle a := x \rangle} \xrightarrow{(b)} (a) \langle a := (a) \langle a := x \rangle \rangle \\
(\lambda a. a) \boxed{(a) \langle a := x \rangle} \xrightarrow{(xv)} (\lambda a. a) x \quad \boxed{((b) \langle b := a \rangle) \langle a := x \rangle} \xrightarrow{(xv)} (a) \langle a := x \rangle \\
\boxed{(\lambda x. x) x} \xrightarrow{(b)} (a) \langle a := x \rangle \quad \boxed{(a) \langle a := (a) \langle a := x \rangle \rangle} \xrightarrow{(xv)} (a) \langle a := x \rangle \\
(a) \langle a := \boxed{(a) \langle a := x \rangle} \rangle \xrightarrow{(xv)} (a) \langle a := x \rangle \quad \boxed{(\lambda a. a) x} \xrightarrow{(b)} (a) \langle a := x \rangle \quad \boxed{(a) \langle a := x \rangle} \xrightarrow{(xv)} x \\
\boxed{(a) \langle a := x \rangle} \xrightarrow{(xv)} x
\end{array}$$

### A.2.3 Example.

Another interesting example is Mellès's (1995) counterexample discussed in Remark 3.2.1. We first resolve the I combinators (notice the ()s around I to ensure it is not seen as a constructor with an argument).

```
? tryC "ski" "'lx.('ly.(I)(Iy))(Ix)"
1 'lx.('ly.('la.a)('la.a)y)('la.a)x
2 (21156 reductions, 66066 cells)
```

Next we search through all reductions of this term.

```
? load"b x gc"
? search "'lx.('ly.('la.a)('la.a)y)('la.a)x"
3 %%0.2
4 %% 0/1 {000:b}
5 %% 0/1 {0000000:b}
6 %%1.4
7 %% 1/1 {000:xap}
8 %% 1/1 {0000000:b}
9 %% 1/1 {000:b}
10 %% 1/1 {0000000:xv}
11 %%2.8
...
12 %% 7/1 {00000:gc}
13 %% 7/1 {000:b}
```

```

14 %% 7/1 {00:xv}
15 %%8.28
16 'lx.x
17 %% 8/1 {000:b}
18 %% 8/1 {000000:xvgc}
19 %% 8/1 {000000:gc}

```

...

```

20 %% 12/1 {00:xv}
21 %%13.1
22 %% 13/1 {00:xv}
23 %%14.0

```

```

24 (495600 reductions, 1130576 cells, 1 garbage collection)

```

(output abbreviated). We observe that the shortest path to the normal form is 7 reductions and the longest path is 14 reductions.

#### A.2.4 Example.

Finally, we derive the critical pairs investigated by the proof of Proposition 3.1.10.c.

```
? crits""
```

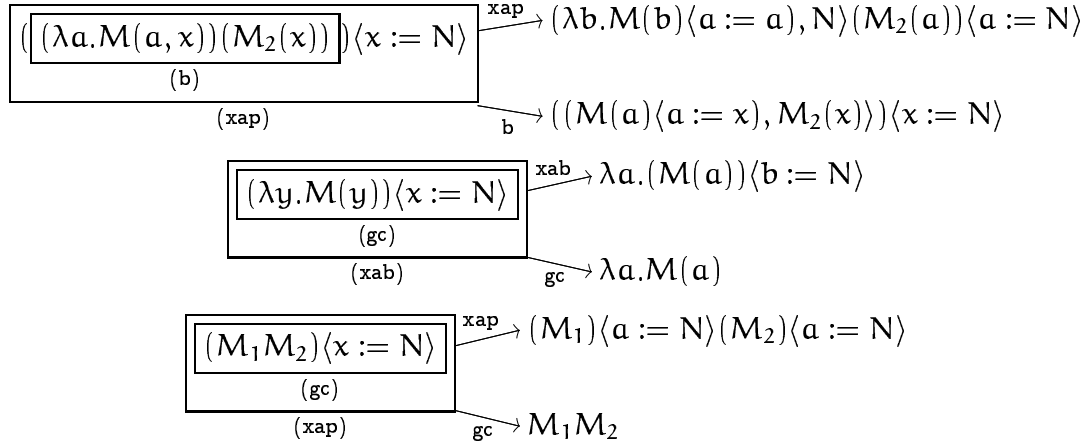
```

1 %CRS? crits""
2 rules xap---b
3 generate %% 0/1 {:xap}
4 \XS ([a]'lb.\mS{M}(b,a),\m{N})\XS ([a]\mS{M}_2(a),\m{N})
5 <-> %% 0/1 {00:b}
6 \XS ([x]\XS ([a]\mS{M}(a,x),\mS{M}_2(x)),\m{N})
7 rules xab---gc
8 generate %% 0/1 {:xab}
9 'la.\XS ([b]\mS{M}(a),\m{N})
10 <-> %% 0/1 {:gc}
11 'la.\mS{M}(a)
12 rules xap---gc
13 generate %% 0/1 {:xap}
14 \XS ([a]\mS{M}_1,\m{N})\XS ([a]\mS{M}_2,\m{N})
15 <-> %% 0/1 {:gc}
16 \mS{M}_1\mS{M}_2
17 (64512 reductions, 135561 cells)

```

The pretty-printed version is shown below.





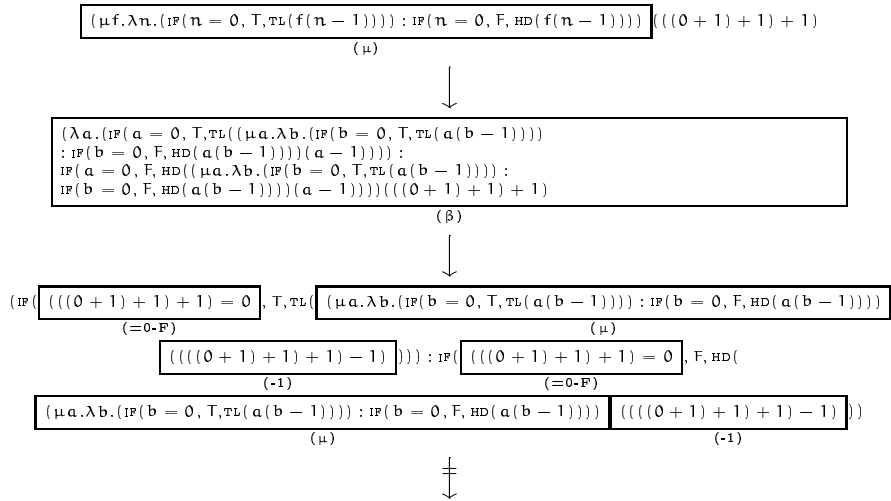
## A.3 PCF

This demonstrates the result of running the CRS system of chapter 6 using variants of PCF+pairs of Plotkin's (1977) PCF language shown in Figure 5.2.

**A.3.1 Example (PCF+pairs).** The full reduction sequence of the example first in section 5.3 using Plotkin's PCF with pairs shown in Figure 5.2 on the term

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

is as follows:





$$\begin{array}{c}
\boxed{\text{HD}((\text{IF}((0+1) = 0, \text{T, TL}((\mu a. \lambda b. (\text{IF}(b = 0, \text{T, TL}(a(b-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))((0+1) - 1)))) : \text{IF}((0+1) = 0, \text{F, HD}((\mu a. \lambda b. (\text{IF}(b = 0, \text{T, TL}(a(b-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))((0+1) - 1))))} \\
\text{(hd)} \\
\boxed{\text{TL}((\text{IF}((0+1) = 0, \text{T, TL}((\mu a. \lambda b. (\text{IF}(b = 0, \text{T, TL}(a(b-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))((0+1) - 1)))) : \text{IF}((0+1) = 0, \text{F, HD}((\mu a. \lambda b. (\text{IF}(b = 0, \text{T, TL}(a(b-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))((0+1) - 1))))} \\
\text{(tl)} \\
\Downarrow \\
(\text{IF}(\boxed{(0+1) = 0} \text{), T, TL}(\boxed{(\mu a. \lambda b. (\text{IF}(b = 0, \text{T, TL}(a(b-1)))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))})) \\
\text{(=0-F)} \quad (\mu) \\
\boxed{((0+1) - 1)} : \boxed{(0+1) = 0} \text{, F, HD}(\boxed{(\mu a. \lambda b. (\text{IF}(b = 0, \text{T, TL}(a(b-1)))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))} \boxed{((0+1) - 1)}) \\
\text{(-1)} \quad \text{(=0-F)} \quad (\mu) \quad \text{(-1)} \\
\Downarrow \\
\boxed{(\text{IF}(\text{F, T, TL}((\lambda a. (\text{IF}(a = 0, \text{T, TL}((\mu b. \lambda c. (\text{IF}(c = 0, \text{T, TL}(b(c-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))(a-1)))) : \text{IF}(a = 0, \text{F, HD}((\mu b. \lambda c. (\text{IF}(c = 0, \text{T, TL}(b(c-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))(a-1))))))} \\
\text{(if-F)} \\
\boxed{(\text{IF}(\text{F, F, HD}((\lambda a. (\text{IF}(a = 0, \text{T, TL}((\mu b. \lambda c. (\text{IF}(c = 0, \text{T, TL}(b(c-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))(a-1)))) : \text{IF}(a = 0, \text{F, HD}((\mu b. \lambda c. (\text{IF}(c = 0, \text{T, TL}(b(c-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))(a-1))))))} \\
\text{(if-F)} \\
\Downarrow \\
(\text{TL}(\boxed{(\lambda a. (\text{IF}(a = 0, \text{T, TL}((\mu b. \lambda c. (\text{IF}(c = 0, \text{T, TL}(b(c-1)))))) : \text{IF}(a = 0, \text{F, HD}((\mu b. \lambda c. (\text{IF}(c = 0, \text{T, TL}(b(c-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))(a-1))))))} : \text{HD}(\boxed{(\text{IF}(c = 0, \text{F, HD}(b(c-1))))(a-1))))} \\
\text{(}\beta\text{)} \\
\boxed{(\lambda a. (\text{IF}(a = 0, \text{T, TL}((\mu b. \lambda c. (\text{IF}(c = 0, \text{T, TL}(b(c-1)))))) : \text{IF}(a = 0, \text{F, HD}((\mu b. \lambda c. (\text{IF}(c = 0, \text{T, TL}(b(c-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))(a-1))))} \\
\text{(}\beta\text{)} \\
\Downarrow \\
\boxed{(\text{TL}((\text{IF}(0 = 0, \text{T, TL}((\mu a. \lambda b. (\text{IF}(b = 0, \text{T, TL}(a(b-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))(0-1)))) : \text{IF}(0 = 0, \text{F, HD}((\mu a. \lambda b. (\text{IF}(b = 0, \text{T, TL}(a(b-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))(0-1))))} \\
\text{(tl)} \\
\boxed{\text{HD}((\text{IF}(0 = 0, \text{T, TL}((\mu a. \lambda b. (\text{IF}(b = 0, \text{T, TL}(a(b-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))(0-1)))) : \text{IF}(0 = 0, \text{F, HD}((\mu a. \lambda b. (\text{IF}(b = 0, \text{T, TL}(a(b-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))(0-1))))} \\
\text{(hd)} \\
\Downarrow \\
(\text{IF}(\boxed{0 = 0} \text{), F, HD}(\boxed{(\mu a. \lambda b. (\text{IF}(b = 0, \text{T, TL}(a(b-1)))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))} \boxed{(0-1)})) : \text{IF}(\boxed{0 = 0} \text{), T, TL}(\boxed{(\mu a. \lambda b. (\text{IF}(b = 0, \text{T, TL}(a(b-1)))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))} \boxed{(0-1)})) \\
\text{(=0-T)} \quad (\mu) \quad \text{(=0-T)} \\
\Downarrow \\
\boxed{(\text{IF}(\text{T, F, HD}((\lambda a. (\text{IF}(a = 0, \text{T, TL}((\mu b. \lambda c. (\text{IF}(c = 0, \text{T, TL}(b(c-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))(a-1)))) : \text{IF}(a = 0, \text{F, HD}((\mu b. \lambda c. (\text{IF}(c = 0, \text{T, TL}(b(c-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))(a-1))))))} \\
\text{(if-T)} \\
\boxed{(\text{IF}(\text{T, T, TL}((\lambda a. (\text{IF}(a = 0, \text{T, TL}((\mu b. \lambda c. (\text{IF}(c = 0, \text{T, TL}(b(c-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))(a-1)))) : \text{IF}(a = 0, \text{F, HD}((\mu b. \lambda c. (\text{IF}(c = 0, \text{T, TL}(b(c-1)))))) : \text{IF}(b = 0, \text{F, HD}(a(b-1))))(a-1))))))} \\
\text{(if-T)} \\
\Downarrow \\
\text{F : T}
\end{array}$$

**A.3.2 Example (PCF with explicit sharing).** Next we show the reduction of the even-odd problem of above but with the sharing PCF CRSar shown in Figure 5.3. The dialogue with the HUGS interpreter was as follows (after a header similar to that of Example A.1.1).

```
? load"pcf.a"
```

```
1 (44949 reductions, 134715 cells, 1 garbage collection)
```

The input is (entered all on one line).

```
? pptryS leftmost
```

```
"('mf. 'ln. \c{if}(n=0)\c{then}(T)\c{else}(\c{tl}(f(n-1)))
      :\c{if}(n=0)\c{then}(F)\c{else}(\c{hd}(f(n-1)))
      (((0+1)+1)+1))"
```

... *output typeset below* ...

```
2 (1730807 reductions, 3342279 cells, 4 garbage collections)
```

The output occupies from here to page 224.

$$\begin{aligned}
 & \frac{(\mu f. \lambda n. (\text{IF}(n = 0, T, \text{TL}(f(n-1)))) : \text{IF}(n = 0, F, \text{HD}(f(n-1))))}{(\mu)} (((0+1)+1) + \\
 1) & \longrightarrow \frac{(\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\blacksquare)^1)(a-1)))) : \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1}{(\text{copy})} (((0+1)+1) + \\
 & \frac{(\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\blacksquare)^1)(a-1)))) : \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)(a-1)))) : \\
 & \text{IF}(a = 0, F, \text{HD}(((\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\blacksquare)^1)(a-1)))) : \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)(a-1)))) : \\
 & \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)((0+1)+1) + 1}{(\beta)} \longrightarrow (\text{IF}(((0+1)+1)+1)^3) \\
 & = 0, T, \text{TL}(((\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\blacksquare)^1)(a-1)))) : \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)((0+1)+1) + 1)^3 - 1)) : \\
 & \text{IF}(((0+1)+1)+1)^3 = 0, F, \text{HD}(((\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\blacksquare)^1)(a-1)))) : \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1) : \\
 & \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)((0+1)+1) + 1)^3 - 1)) \longrightarrow (\text{IF}(((0+1)+1) = 0) \\
 & , T, \text{TL}(((\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\blacksquare)^1)(a-1)))) : \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)((0+1)+1) + 1)^3 - 1)) : \\
 & \text{IF}(((0+1)+1)+1)^3 = 0, F, \text{HD}(((\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\blacksquare)^1)(a-1)))) : \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)((0+1)+1) + \\
 1) & \frac{(\text{IF}(F, T, \text{TL}(((\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\blacksquare)^1)(a-1)))) : \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1) : \\
 & \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)((0+1)+1) + 1)^3 - 1))}{(\text{if-F})} : \\
 & \text{IF}(((0+1)+1)+1)^3 = 0, F, \text{HD}(((\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\blacksquare)^1)(a-1)))) : \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)((0+1)+1) + \\
 1) & \frac{(\text{TL}(((\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\blacksquare)^1)(a-1)))) : \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)((0+1)+1) + 1)^3 - 1))}{(\text{copy})} : \\
 & \text{IF}(((0+1)+1)+1)^3 = 0, F, \text{HD}(((\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\blacksquare)^1)(a-1)))) : \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)((0+1)+1) + \\
 1) & \frac{(\text{TL}(((\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\blacksquare)^1)(a-1)))) : \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)(a-1)))) : \\
 & \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)(a-1)))) : \\
 & \text{IF}(a = 0, F, \text{HD}(((\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\blacksquare)^1)(a-1)))) : \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)(a-1)))) : \\
 & \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)((0+1)+1) + 1)^3 - 1)}{(\beta)} : \\
 & \text{IF}(((0+1)+1)+1)^3 = 0, F, \text{HD}(((\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\blacksquare)^1)(a-1)))) : \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)((0+1)+1) + \\
 1) & \frac{(\text{TL}(((\text{IF}(((0+1)+1)+1)^3 - 1)^8 = 0, T, \text{TL}(((\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\blacksquare)^1)(a-1)))) : \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1) : \\
 & \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)((0+1)+1) + 1)^3 - 1)^8 - 1)) : \\
 & \text{IF}(((0+1)+1)+1)^3 - 1)^8 = 0, F, \text{HD}(((\lambda a. (\text{IF}(a = 0, T, \text{TL}(((\blacksquare)^1)(a-1)))) : \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1) : \\
 & \text{IF}(a = 0, F, \text{HD}(((\blacksquare)^1)(a-1))))^1)((0+1)+1) + 1)^3 - 1)^8 - 1))}{(\text{tl})} :
 \end{aligned}$$











**A.3.3 Code (PCF with explicit sharing and substitution).** The ‘real complexity’ of the reduction of Example A.3.2 above is shown by the result of running the same reduction in the explicified system which is shown in figure Figure 5.4.

```
? load(expl"pcfa")
3 (325175 reductions, 722552 cells, 1 garbage collection)
```

The reduction shows what happens: some ‘shared’ redexes are reduced that cannot be reduced.

```
? tryS leftmost
  "(('mf.'ln.\c{if}(n=0)\c{then}(T)\c{else}(\c{t1}(f(n-1)))
    :\c{if}(n=0)\c{then}(F)\c{else}(\c{hd}(f(n-1))))
  (((0+1)+1)+1)"
```

The output is as follows – for brevity only an excerpt of the list of redexes is shown, not the actual terms.

```
4 %% 1/1 {0:$'m$-x}
5 %% 2/1 {0:copy}
6 %% 3/1 {0:x-$'1$-$1$}
7 %% 4/1 {00:xma-$1$}
8 %% 5/1 {:$'b$-x}
9 %% 6/1 {00:x-$-$1$}
10 %% 7/1 {:x-$-$1$}
11 %% 8/1 {000:x-$0$-$1$}
12 %% 9/1 {0:x-$0$-$1$}

...

13 %% 332/1 {10000:-1}
14 %% 333/1 {10001:xgc-$1$-$1$}
15 %% 334/1 {10001:x-$0$-$1$}
16 %% 335/1 {1000:=0-T}
17 %% 336/1 {101:xgc-$1$-$1$}
18 %% 337/1 {101:x-$\lambda c\{then\}$-$1$}
19 %% 338/1 {1010:x-$T$-$1$}
20 %% 339/1 {1100:x-$\lambda c\{else\}$-$1$}
21 %% 340/1 {11:x-$\lambda c\{else\}$-$1$}
22 %% 341/1 {1:if-T}

23 F:T

24 (4199996 reductions, 9141399 cells, 11 garbage collections)
```

## A.4 Pretty-printing Metaterms with Redexes

This section presents a complex pretty-printer that prints a metaterm and a set of redexes.<sup>2</sup> It is a generalisation of `showMterm` from Code 6.2.10.

<sup>2</sup>The section is the result of typesetting the literate Haskell script “`crsprettys.lhs`”.

### A.4.1 Code.

```
1 rxMterm,rxMterm' :: [Redex] → Mterm → String
```

First we shave off all the irrelevant stuff of the redexes, leaving just a list of pairs of paths and rule names.

```
2 rxMterm = rxMterm'' False
3 rxMterm' = rxMterm'' True
4 rxMterm'' b rxs t = (case str of
5     "" → ""
6     ws → "\\Metaterm{\n"+ws+"\\n}%\n") where
7   str = brk (words (showMterm' t [ (λ(Rx p _ (_,_,nm)) → (p,nm))rx | rx ← rxs ]))
8   brk [] = ""
9   brk (w:ws) = w++brk' (balance w) (length w) ws
10  brk' i _ [] = ""
11  brk' i n (w:ws) | n_lw < brklength = "␣"+w++brk' i' n_lw ws
12                  | b = "\\break{\n"+indent i++w++brk' i' lw ws
13                  | otherwise = "\\allowbreak\n"+w++brk' i' lw ws
14                  where lw = strlength w ; n_lw = n+lw+1
15                          i' = i+balance w
16  brklength = 80
17  strlength w = length (filter ('notElem' "\\{}␣") w)
18  balance w = length (filter ('(' ≡) w) - length (filter (')' ≡) w)
19  indent 0 = ""
20  indent (i+1) = '~':indent i
```

Two auxiliaries are essential. The first prints out appropriate marks for any redexes at just the current point.

```
21 p :: [(Path,String)] → String → String
22 p rs s = p2 rs s (p1 rs)
23 p1 rs = [ nm | (ns,nm) ← rs, ns ≡ [] ]
24 p2 rs s here = foldr markredex s here where
25   markredex nm s | length s < 200 = "␣\\markredex{"++s++"}{"++nm++"}␣"
26                   | otherwise = "␣\\markredexlong{"++s++"}{"++nm++"}␣"
```

The second filters out just those redexes that are local to the *n*th subterm.

```
27 q :: Int → [(Path,String)] → [(Path,String)]
28 q n rs = below
29   where
```

```

30   below = [ (tail ns,nm) | (ns,nm) ← rs, current(ns,nm) ]
31   current ((n':ns'),nm) | n' ≡ n      = True
32                               | otherwise = False
33   current (_,_)                = False

```

Some shorthands:

```

34   q0 = q 0
35   q1 = q 1

```

The rest is just the `showMterm` local definitions except that all the functions have an extra redexes argument `r`. The rule for `p` is: “When ever term structure is stripped off, make sure to mark that redex.” The rule for `q` is to apply it such that “Whenever a term is passed, `r` is local to it. Whenever a list of terms is passed, `r` is local to the parent.” The only remaining issue is that the functions marked `INNER` do not print redexes because they are called at points where printing of redexes has already happened.

This function is reduced in size because it is not reduced in complexity.

```

36 showMterm' (Mvar v)      r = p r v
37 showMterm' (Mabs v t)   r = p r (showMabs v t (q0 r))
38 showMterm' (Mcon fs ts) r = p r (showMcon fs ts r)
39 showMterm' (Mapp mv ts) r = p r (showSym mv ++ showMtermlist ts r)
40 showMabs vs (Mabs v t) r | p1 r ≡ [] = showMabs (vs++v) t (q0 r)          --INNER
41                               | otherwise = "[" ++ vs ++ "]" ++ showMterm' (Mabs v t) r
42 showMabs vs t           r           = "[" ++ vs ++ "]" ++ showMterm' t r
43 showMcon fs@(s,1,False) [Mabs v t] r = showMcon' fs v t (q0 r)          --INNER
44 showMcon fs@(s@('':_),2,True) [t1,t2] r =
45   "⌊{" ++ showSimple' t1(q0 r) ++ "⌋" ++ s ++ "⌋" ++ showSimple' t2(q1 r) ++ "⌋"⌋"
46 showMcon fs@(":",2,True) [t1,t2] r = showSimple t1(q0 r) ++ "⌋:⌋" ++ showMterm' t2(q1 r)
47 showMcon fs@("@",2,_) [s,t] r = showApp s t r
48 showMcon fs@("^",2,True) [s,t] r = "(" ++ showMterm' s(q0 r) ++ ")" ^ showSimple t(q1 r) ++ "\\,"
49 showMcon fs@(s,2,True) [t1,t2] r = showSimple t1(q0 r) ++ s ++ showSimple t2(q1 r)
50 showMcon fs@("{,}",0,False) [] r = "()"
51 showMcon fs@("{,}",n+2,False) ts r = showMtermlist ts r
52 showMcon fs@("<>",0,False) [] r = "<>"
53 showMcon fs@("<>",n+1,False) ts r = "<" ++ showMtermlist' ts r 0 ++ ">"
54 showMcon fs@(s@('\'':_):_),False) ts r =
55   showSym fs ++ "⌊{" ++ showMtermlist' ts r 0 ++ "⌋"⌋"
56 showMcon fs
57   ts r = showSym fs ++ showMtermlist ts r
58 showMcon' fs@(s,1,False) vs t@(Mcon (s',1,False) [Mabs v t]) r          --INNER
59   | s ≡ s' ∧ p1 r ≡ [] = showMcon' fs (vs++v) t (q0 r)
60   | otherwise = showSym fs ++ vs ++ "." ++ showMterm' t'(q0 r)
61 showMcon' fs@("@",2,_) ts r = "\\(" ++ showMcon fs ts r ++ "\\)" --INNER
62 showMcon' fs@(s,1,False) [Mabs v t] r = "\\(" ++ showMcon' fs v t (q0 r) ++ "\\)"
63 showMcon' fs@(_,a,False) ts r | a ≡ 0 = "\\(" ++ showMcon fs ts r ++ "\\)"
64                               | otherwise = showMcon fs ts r
65 showMcon' fs ts r = "\\(" ++ showMcon fs ts r ++ "\\)"
66 showSimple' (Mcon (s@('':_),2,True) [t1,t2]) r =
67   p r ("⌊{" ++ showSimple t1(q0 r) ++ "⌋" ++ s ++ "⌋" ++ showSimple t2(q1 r) ++ "⌋"⌋")
68 showSimple' t r = showSimple t r
69 showSimple' (Mvar v) r = p r v

```

```

70 showSimple (Mabs v t)                r = p r ("\\("++showMabs v t(q0 r)+"\\)")
71 showSimple (Mcon fs [])              r = p r (showSym fs)
72 showSimple (Mcon (s@('':_){':_},2,True) [t1,t2]) r =
73   p r ("\\<"++showMterm' t1(q0 r)+"_++s+"_++showMterm' t2(q1 r)+"\\>")
74 showSimple (Mcon ("@",2,_) [t1,t2]) r = p r (showApp t1 t2 r)
75 showSimple (Mcon ("^",2,True) [s,t]) r = p r ("("++showMterm' s(q0 r)+"~"++showSimple t(q1 r)+"\\,")
76 showSimple (Mcon ("{,}"",0,False) []) r = p r "(")
77 showSimple (Mcon ("[,]"",n+2,False) ts) r = p r (showMtermlist ts r)
78 showSimple (Mcon ("<>",0,False) []) r = p r "<>"
79 showSimple (Mcon ("<>",n,False) ts) r = p r ("<"++showMtermlist' ts r 0++">")
80 showSimple (Mcon fs ts)              r = p r ("\\("++showMcon fs ts r++"\\)")
81 showSimple (Mapp mv [])              r = p r (showSym mv)
82 showSimple (Mapp mv ts) r = p r ("\\("++showSym mv++showMtermlist ts r++"\\)")

83 showApp s (Mvar v) r = showB4var s(q0 r)+p(q1 r)v --INNER
84 showApp s (Mcon fs []) r = showSimple s(q0 r)+p(q1 r)(showSym fs)
85 showApp s t@(Mcon fs ts) r = showB4par s(q0 r)+p(q1 r)(showMcon' fs ts (q1 r))
86 showApp s (Mapp mv []) r = showSimple s(q0 r)+p(q1 r)(showSym mv)
87 showApp s t r = showB4par s(q0 r)+"\\("++showMterm' t(q1 r)+"\\)"

88 showB4par (Mvar v) r = p r v
89 showB4par (Mcon ("@",2,_) [s,Mvar v]) r = p r (showB4var s(q0 r)+p(q1 r)v)
90 showB4par t@(Mcon fs@(s,a,False) ts) r = p r (showMcon' fs ts r)
91 showB4par t r = "\\("++showMterm' t r++"\\)"

92 showB4var (Mvar v) r = p r v
93 showB4var (Mapp mv []) r = p r (showSym mv)
94 showB4var (Mcon fs []) r = p r (showSym fs)
95 showB4var (Mcon ("@",2,_) [s,Mvar v]) r = p r (showB4var s(q0 r)+p(q1 r)v)
96 showB4var t r = "\\("++showMterm' t r++"\\)"

97 showMtermlist [] r = "" --INNER
98 showMtermlist ts r = "\\("++showMtermlist' ts r 0++"\\)"

99 showMtermlist' [t] r n = showMterm' t(q n r) --INNER+INDUCTIVE
100 showMtermlist' (t:ts) r n = showMterm' t(q n r)+"",++showMtermlist' ts r (n+1)

```

**A.4.2 Code (print reduction).** Following are versions of the `nf` and `nfs` functions of section 6.5 that do extensive printing using the above print function. First the pretty version of `nf`.

```

1  ppnf,ppnf' :: CRS -> CRS -> Strategy -> Mterm -> String
2  ppnf = ppnf'' False
3  ppnf' = ppnf'' True
4  ppnf'' b [] [] _ t = rxMterm'' b [] t
5  ppnf'' b crs crs2 strat t = nf' 0 t where
6    nf' n t | rxs ≡ [] = rxMterm'' b [] t'
7             | otherwise = rxMterm'' b rxs t'
8             ++ "\\Reduction{\\->!{^"++shownumb n++"}\\}%\n"
9             ++nf' n1 (rewrite n1 rxs t')
10         where rxs = strat t' (redexes crs t')
11               n1 = n+1
12               t' = nf crs2 [] outermost t

```

And the pretty `nfs`.

```

13 ppnfs,ppnfs' :: CRS → CRS → Strategy → Mterm → String
14 ppnfs = ppnfs'' False
15 ppnfs' = ppnfs'' True
16 ppnfs'' b [] [] _ t = rxMterm [] t
17 ppnfs'' b crs crs2 strat t = snd(nfs' 0 [] [t]) where
18   nfs' :: Int → [Mterm] → [Mterm] → ([Mterm],String)
19   nfs' n _ [] = ([],"")
20   nfs' n old current = (newnfs
21                         ++tr (show n1++"/"++show (length current))
22                             terms',
23                         reds'
24                         ++unlines newreds
25                         ++unlines [ rxMterm'' b [] t | t ← newnfs])
26   where
27     (terms',reds') = nfs' n1 (old++newnfs) new
28     n1 = n+1
29     newnfs = [ t | (t,_) ← nfpert, t 'notElem' old ]
30     (new,newreds) = unzip [ tt rx t (rewrite n1 [rx] t)
31                           | (t,rxs) ← nonnfpert, rx ← rxs ]
32     tt rx t t' = (t', rxMterm'' b [rx] t+"\\Reduction{\"->!\"}%\n"
33                 ++rxMterm [] t')
34     (nfpert,nonnfpert) =
35       partition (λ(_,rx) → rx ≡ [])
36       [ (t, strat t (redexes crs t)) | t ← current' ]
37     current' = map (nf crs2 [] outermost) (nub current)

```

### A.4.3 Code (pretty interaction). Noisy version of the interactive functions in section 6.9.

```

38 ppstep s = rxMterm[] (onestep (crsf "crs1") (crsf "crs2") outermost (term s))
39 pptry s = ppnf (crsf "crs1") (crsf "crs2") outermost (term s)
40 pptryi s = ppnf (crsf "crs1") (crsf "crs2") innermost (term s)
41 ppsearch s = ppnfs (crsf "crs1") (crsf "crs2") strong (term s)
42 ppstepS strat s = rxMterm[] (onestep (crsf "crs1") (crsf "crs2") strat (term s))
43 pptryS strat s = ppnf (crsf "crs1") (crsf "crs2") strat (term s)
44 ppsearchS strat s = ppnfs (crsf "crs1") (crsf "crs2") strat (term s)

```

Alternate versions of some for longer reductions.

```

45 pptry' s = ppnf (crsf "crs1") (crsf "crs2") outermost (term s)
46 pptryi' s = ppnf (crsf "crs1") (crsf "crs2") innermost (term s)
47 ppsearch' s = ppnfs' (crsf "crs1") (crsf "crs2") strong (term s)

```

```

48 ppstepS' strat s = rxMterm' [] (onestep (crsf "crs1") (crsf "crs2") strat (term s))
49 pptryS' strat s = ppnf' (crsf "crs1") (crsf "crs2") strat (term s)
50 ppsearchS' strat s = ppnfs' (crsf "crs1") (crsf "crs2") strat (term s)

```

And the ‘driver’ function that delays the actual normal forms to last.

```

51 delay ts = delay' ts ts where
52   delay' []      ts' = show ts'
53   delay' (t:ts) ts' = checkMterm t ++ delay' ts ts'

```

#### A.4.4 Code (overlaps). Printing overlaps and critical pairs.

```

54 pplaps :: String → String
55 pplaps c = "\n" ++ foldr1 (\x y → x ++ "\\\n" ++ y)
56   [ rxMterm [rx1,rx2] t | (t,rx1,rx2) ← overlaps (crs c) ]

57 ppcrits :: String → String
58 ppcrits c = "\n"
59 ++ case [ ppcritwithreducts t rx1 rx2 (rewrite 0[rx1]t) (rewrite 0[rx2]t)
60         | (t,rx1,rx2) ← overlaps (crs c) ]
61   of [] → "\\text{No critical pairs!}"
62      cps → foldr1 (\x y → x ++ "\\\n" ++ y) cps
63 ++ "\n"

```

We show a critical pair and two reducts using Xy-pic.

```

64 ppcritwithreducts t rx1@(Rx _ _ (_,_,nm1)) rx2@(Rx _ _ (_,_,nm2)) t1 t2 =
65   "\\t{\\vcenter{\\xy0*{\\displaystyle_} ++ rxMterm [rx1,rx2] t ++ \"t} = \" s \"
66   ++ \"\\ar_+R+<2pc,+1.5pc>+!L{\\displaystyle_}
67   ++ rxMterm [t1 ++ \"t} - {\\text{\" ++ nm1 ++ \"}}\"
68   ++ \"\\ar_+R+<2pc,-1.5pc>+!L{\\displaystyle_}
69   ++ rxMterm [t2 ++ \"t} - {\\text{\" ++ nm2 ++ \"}}\"
70   ++ \"\\endxy}}\"

```

#### A.4.5 Code (local confluence). Testing that the normal forms of locally divergent reductions converge.

```

1  pplc :: String → Check
2  pplc c =
3    either "\\text{CRS not locally confluent!}\\\\n"
4          (foldr1 (\x y → x ++ "\\\n" ++ y)
5            [ ppcritwithreducts t rx1 rx2 t1 t2
6              | (t,rx1,rx2) ← overlaps c',
7                t1 ← nfs c' [] strong (rewrite 0 [rx1] t),
8                t2 ← nfs c' [] strong (rewrite 0 [rx2] t),
9                t1 ≠ t2 ])
10   where c' = crs c

```

#### A.4.6 Code (sample output). These prefix commands provide quick&dirty output to test files.

```

11 use c = writeFile "sample.done" "Updating CRSs\n" complain (load c)
12 use2 c c2 = writeFile "sample.done" "Updating CRSs\n" complain (load2 c c2)

```

```
13 run f = sr (pptry (show (termf f)))
14 runS strat f = sr (pptryS strat (show (termf f)))
15 runsearch f = sr (ppsearch (show (termf f)))

16 runt f = sr (pptry (show (term f)))
17 runtS strat f = sr (pptryS strat (show (term f)))
18 runsearcht f = sr (ppsearch (show (term f)))

19 test f = sc (f "")

20 sr s = sx "tex" ("\\begin{center}\\CRScenter" ++ s ++ "\\end{center}\\n")
21 sc s = sx "tex" ("{\\CRScrits\\begin{gather*}" ++ s ++ "\\end{gather*}}\\n")

22 sx x s = writeFile ("sample."+x) s complain $
23     writeFile "sample.done" ("Updating\\CRSs\\(" ++ x ++ "\\component)\\n") complain $
24     done
```





# Bibliography

- Abadi, M., Cardelli, L., Curien, P.-L. and Lévy, J.-J. (1991). Explicit substitutions. *Journal of Functional Programming* 1(4): 375–416.
- Abramsky, S. (1990). The lazy lambda calculus. In Turner, D. A. (ed.), *Research Topics in Functional Programming*. Addison-Wesley. Chapter 4: pp. 65–116.
- Aczel, P. (1978). A general Church-Rosser theorem. *Technical report*. Univ. of Manchester.
- Aho, A. V., Sethi, R. and Ullman, J. D. (1986). *Compilers: Principles, Techniques and Tools*. Addison-Wesley.
- American Mathematical Society (1995).  *$\mathcal{A}\mathcal{M}\mathcal{S}$ - $\mathcal{L}\mathcal{T}\mathcal{E}\mathcal{X}$  Version 1.2 User's Guide*. [URL: ftp://ftp.tex.ac.uk/ctan/tex-archive/macros/ams/amslatex/](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.
- Ariola, Z. M. and Klop, J. W. (1994). Cyclic lambda graph rewriting. *LICS '94—Ninth Annual IEEE Symposium on Logic in Computer Science*. Paris, France. pp. 416–425. full version forthcoming (Ariola and Klop 1995b).
- Ariola, Z. M. and Klop, J. W. (1995a). Equational term graph rewriting. *Technical report*. CWI. To appear in *Acta Informatica*.

- Ariola, Z. M. and Klop, J. W. (1995b). Lambda calculus with explicit recursion. Personal Communication.
- Backus, J. (1978). Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM* 21. Based on 1977 Turing lecture.
- Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics*. Revised edn. North-Holland.
- 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.
- Benaïssa, Z.-E.-A., Briaud, D., Lescanne, P. and Rouyer-Degli, J. (1995).  $\lambda\nu$ , 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.  $\langle$ URL: <http://www.loria.fr/~lescanne/PUBLICATIONS/RR-2477.PS> $\rangle$
- Benaïssa, Z.-E.-A. and Lescanne, P. (1995). Triad machine: A general computation model for the description of abstract machines. *Technical Report 95-R-410*. CRIN (Centre de Recherche en Information de Nancy), Bâtiment Loria. B.P.239, F-54506 Vandoeuvre lès Nancy Cedex.
- Berners-Lee, T., Fielding, R. and Frystyk, H. (1995). Hypertext transfer protocol – HTTP/1.0. IETF Internet Draft.  $\langle$ URL: <http://www.ics.uci.edu/pub/ietf/http/draft-ietf-http-v10-spec-03.html> $\rangle$
- Bird, R. and Wadler, P. (1988). *Introduction to Functional Programming*. Prentice-Hall.
- 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.
- 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⟩](ftp://ftp.diku.dk/diku/semantics/papers/D-246.ps)

- 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*. Rutgers University. Springer-Verlag. New Brunswick, New Jersey. pp. 169–183. [⟨URL: http://www.brics.dk/~krisrose/PAPERS/bloo+rose-rta96.ps.gz⟩](http://www.brics.dk/~krisrose/PAPERS/bloo+rose-rta96.ps.gz)
- 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).
- 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.
- Cailliau, R. (1995). A little history. World Wide Web. [⟨URL: http://www.w3.org/pub/WWW/History.html⟩](http://www.w3.org/pub/WWW/History.html)
- Church, A. (1936). A note on the entscheidungsproblem. *J. Symbolic logic* 58: 354–363.
- Church, A. (1941). *The Calculi of Lambda-Conversion*. Princeton University Press. Princeton, N. J.
- Church, A. and Rosser, J. B. (1935). Some properties of conversion. *Transactions on the AMS* 39: 472–482.
- Claus, V., Ehrig, H. and Rozenberg, G. (eds) (1978). *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.
- Collins (1979). *English Dictionary*. William Collins Sons & Co. Ltd.
- Cousineau, G., Curien, P.-L. and Mauny, M. (1987). The categorical abstract machine. *Science of Computer Programming* 8: 173–202.

- Crégut, P. (1990). An abstract machine for the normalisation of  $\lambda$ -terms. *In* LFP '90 (1990). pp. 333–340.
- Curry, H. B. and Feys, R. (1958). *Combinatory Logic*. Vol. I. North-Holland.
- Danthine, A. A. S. (1980). Protocol representation with finite-state models. *IEEE Trans. on Commun.* COM-28: 632–643.
- 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.
- Ehrig, H. (1978). Introduction to the algebraic theory of graph grammars. *In* Claus, Ehrig and Rozenberg (1978). pp. 1–69.
- Fairbairn, J. and Wray, S. C. (1987). TIM: A simple, lazy abstract machine to execute supercombinators. *In* Kahn, G. (ed.), *FPCA '87—Functional Programming Languages and Computer Architecture*. Number 274 in LNCS. Springer-Verlag. Portland, Oregon. pp. 34–45.
- Felleisen, M. and Friedman, D. P. (1989). A syntactic theory of sequential state. *Theoretical Computer Science* 69: 243–287.
- 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.
- Gödel, K. (1931). über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatsh für Math. u Phys.* 12(XXXVIII): 173–198.
- 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](ftp://ftp.cl.cam.ac.uk/papers/adg/hug93.dvi.gz)
- Grue, K. (1987). Call-by-mix: A reduction strategy for pure  $\lambda$ -calculus. Unpublished note from DIKU (University of Copenhagen).
- Hannan, J. and Miller, D. (1990). From operational semantics to abstract machines: Preliminary results. *In* LFP '90 (1990). pp. 323–332.

- Henderson, P. (1980). *Functional Programming—Application and Implementation*. Prentice-Hall.
- Hudak, P. and Fasel, H. (1992). A gentle introduction to Haskell. *SIGPLAN Notices* 27(5). [⟨URL: ftp://ftp.dcs.gla.ac.uk/pub/haskell/tutorial/tutorial.ps.Z⟩](ftp://ftp.dcs.gla.ac.uk/pub/haskell/tutorial/tutorial.ps.Z)
- Hudak, P., Peyton Jones, S. L., Wadler, P. and others (1992). Report on the programming language Haskell. *SIGPLAN Notices* 27(5). Version 1.2. [⟨URL: ftp://ftp.dcs.gla.ac.uk/pub/haskell/report/report-1.2.ps.Z⟩](ftp://ftp.dcs.gla.ac.uk/pub/haskell/report/report-1.2.ps.Z)
- Huet, G. (1980). Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM* 27(4): 797–821.
- Hughes, J. M. (1982). Super-combinators: A new implementation method for applicative languages. *LFP '82—ACM Symposium on LISP and Functional Programming*. Pittsburgh, Pennsylvania. 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.
- Jones, M. P. (1995). HUGS – Haskell user's gofer system (v1.01). Internet distributed. [⟨URL: ftp://ftp.cs.nott.ac.uk/pub/haskell/hugs⟩](ftp://ftp.cs.nott.ac.uk/pub/haskell/hugs)
- Kahn, G. (1987). Natural semantics. *Rapport 601*. INRIA. Sophia-Antipolis, France.
- Kahrs, S. (1993). Compilation of combinatory reduction systems. Extended version of HOA'93 paper. [⟨URL: ftp://ftp.dcs.ed.ac.uk/pub/smk/CRS/compile.dvi⟩](ftp://ftp.dcs.ed.ac.uk/pub/smk/CRS/compile.dvi)
- 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  $\lambda$ -calculus à la de Bruijn with explicit substitutions. In Hermenegildo, M. and Swierstra, S. D. (eds), *PLILP '95—Seventh International Symposium on Programming Languages:*

*Implementation, Logics and Programs*. Number 982 in *LNCS*. Springer-Verlag. Utrecht, The Netherlands. pp. 45–62.

- 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.
- Kleene, S. C. (1981). Origins of recursive function theory. *Annals of the History of Computing* 3: 52–67.
- Klop, J. W. (1980). *Combinatory Reduction Systems*. PhD thesis. University of Utrecht. Also available as Mathematical Centre Tracts 127.
- 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. Chapter 1: pp. 1–116.
- Klop, J. W., van Oostrom, V. and van Raamsdonk, F. (1993). Combinatory reduction systems: Introduction and survey. *Theoretical Computer Science* 121: 279–308.
- Knuth, D. E. (1973). *Fundamental Algorithms*. Vol. 1 of *The Art of Computer Programming*. second edn. Addison-Wesley.
- Knuth, D. E. (1984). *The T<sub>E</sub>Xbook*. Addison-Wesley.
- Koopman, P. W. M. (1990). *Functional Programs as Executable Specifications*. PhD thesis. University of Nijmegen.
- 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). *L<sup>A</sup>T<sub>E</sub>X—A Document Preparation System*. 2nd edn. Addison-Wesley.
- Landin, P. J. (1964). The mechanical evaluation of expressions. *Computer Journal* 6: 308–320.

- Launchbury, J. (1993). A natural semantics for lazy evaluation. *POPL '93—Twentieth Annual ACM Symposium on Principles of Programming Languages*. Charleston, South Carolina. pp. 144–154.
- Lescanne, P. (1994a). From  $\lambda\sigma$  to  $\lambda\nu$ : a journey through calculi of explicit substitutions. *POPL '94—21st Annual ACM Symposium on Principles of Programming Languages*. Portland, Oregon. pp. 60–69.
- Lescanne, P. (1994b). On termination of one rule rewrite system. *Theoretical Computer Science* 132: 395–401.
- Lescanne, P. and Rouyer-Degli, J. (1995). Explicit substitutions with de Bruijn's levels. In Hsiang, J. (ed.), *RTA '95—Rewriting Techniques and Applications*. Number 914 in *LNCS*. Springer-Verlag. Kaiserslautern, Germany. pp. 294–308.
- Lévy, J.-J. (1978). *Réductions Correctes et Optimales dans le Lambda-Calcul*. Thèse d'état. Université Paris 7.
- LFP '90 (1990). *LFP '90—ACM Conference on LISP and Functional Programming*. Nice, France.
- McCarthy, J. (1960). Recursive functions of symbolic expressions. *Communications of the ACM* 3(4): 184–195.
- 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.
- Melliès, P.-A. (1995). Typed  $\lambda$ -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.
- Mogensen, T. (1993). *Ratatosk – A Parser Generator and Scanner Generator for Gofer*. [URL: ftp://ftp.diku.dk/diku/users/torbenm/Ratatosk.tar.Z](ftp://ftp.diku.dk/diku/users/torbenm/Ratatosk.tar.Z)
- Mosses, P. (1979). Sis – semantics implementation system, reference manual and user's guide. *Technical Report MD-30*. DAIMI, Aarhus University.

- 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).
- Nipkow, T. (1991). Higher-order critical pairs. *LICS '91—Sixth Annual IEEE Symposium on Logic in Computer Science*. 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.
- Peyton Jones, S. L. and Lester, D. (1992). *Implementing Functional Languages*. Prentice-Hall.
- Peyton Jones, S. L. and Salkild, J. (1989). The spineless tagless G-machine. *FPCA '89—Functional Programming Languages and Computer Architecture*. Addison-Wesley. Imperial College, London. pp. 184–201.
- Pisano, L. (1209). *Liber Abbaci*.
- Plasmeijer, M. J. and van Eekelen, M. C. D. J. (1993). *Functional Programming and Parallel Graph Rewriting*. International Computer Science Series. Addison-Wesley.
- 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  $\lambda$ -calculus. *Theoretical Computer Science* 1: 125–159.
- Plotkin, G. D. (1977). LCF considered as a programming language. *Theoretical Computer Science* 5: 223–255.
- Plotkin, G. D. (1981). A structural approach to operational semantics. *Technical Report FN-19*. DAIMI, Aarhus University. Aarhus, Denmark.
- 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.
- Raoult, J. C. (1984). On graph rewritings. *Theoretical Computer Science* 32: 1–24.
- 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](ftp://ftp.diku.dk/diku/semantics/papers/D-270.ps)
- 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](ftp://ftp.diku.dk/diku/semantics/papers/D-143.ps)
- 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](ftp://ftp.diku.dk/diku/semantics/papers/D-146.ps)
- Rose, K. H. (1994). Summary of qsymbols. Common T<sub>E</sub>X archive Network. [URL: ftp://ftp.tex.ac.uk/tex-archive/macros/latex/contrib/supported/qsymbols](ftp://ftp.tex.ac.uk/tex-archive/macros/latex/contrib/supported/qsymbols)

- Rose, K. H. (1995). Combinatory reduction systems with explicit substitution. *HOA '95 – Second International Workshop on Higher-Order Algebra, Logic and Term Rewriting*. Paderborn, Germany. [⟨URL: ftp://ftp.diku.dk/diku/semantics/papers/D-247.ps⟩](ftp://ftp.diku.dk/diku/semantics/papers/D-247.ps)
- Rose, K. H. (1996). *Operational Reduction Models for Functional Programming Languages*. PhD thesis. DIKU (University of Copenhagen). Universitetsparken 1, DK-2100 København Ø. DIKU report 96/1.
- Rose, K. H. and Bloo, R. (1995). Deriving requirements for preservation of strong normalisation in lambda calculi with explicit substitution. Presented at the EC Lambda-Calculus Meeting, Edinburg, Scotland.
- Rose, K. H. and Moore, R. R. (1995). X<sub>y</sub>-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⟩](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.
- 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⟩](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 et al. (1978). 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 Laboratory.
- 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).

- Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.* Vol. 42 of 2. pp. 230–265.
- Turing, A. M. (1937). Computability and  $\lambda$ -definability. *J. Symbolic Logic* 2: 153–163.
- Turner, D. A. (1979). A new implementation technique for applicative languages. *Software Practice and Experience* 9: 31–49.
- Wadsworth, C. P. (1971). *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis. Programming Research Group, Oxford University.
- Wirth, N. (1971). The programming language PASCAL. *Acta Informatica* 1: 35–63.
- Yoshida, N. (1993). Optimal reduction in weak  $\lambda$ -calculus with shared environments. *FPCA '93—Functional Programming Languages and Computer Architecture*. Addison-Wesley. Copenhagen, Denmark.



# Index

This index collects definitions, concepts, people, and symbols discussed in the dissertation. The ordering is strictly alphabetical with symbols ordered according to their English spelling, *e.g.*,  $\beta$  and  $\xrightarrow{\beta}$  are ordered as ‘beta’. Typewriter font is used for entries to program fragments.

## A

- Abadi, M., 11 (Contribution 1.1.1), 63, 233
- Abrahams, P. W., 26, 45, 239
- Abramsky, S., 30, 49 (History 2.4.10), 233, 238
- $|\_|$ , *see* size
- abstract machine, 115 (Definition 4.1.1)
  - derivation, 116 (Principle 4.1.3)
- abstract reduction system, *see* reduction
- abstract rewrite system, *see* reduction
- abstract syntax, 38 (History 2.2.4)
- abstraction, 41 (Definition 2.3.1)
  - CRS, 52 (Definition 2.6.1)
- $\xrightarrow[\text{bxa}]{\text{a}}$ , 129 (Definition 4.4.1)
- acknowledgements, 3
- acyclic  $\lambda$ -graph reduction, 143 (Example 5.2.2)
- Aczel, P., 51, 233
- $\text{addr}(t)$ , 97 (Definition 3.4.1)
- address, 97 (Definition 3.4.1), 111 (Discussion 3.5.8), 174, 180 (Code 6.4.3)
  - CRS, 142 (Definition 5.2.1)
  - fresh, 142 (Definition 5.2.1)
  - oracle, 143, 174, 180 (Code 6.4.3)
- addresses ( $\text{addr}(\_)$ ), 97 (Definition 3.4.1)
- admissible, 102 (Definition 3.4.9), 104 (History 3.4.13)
  - local, 109 (Definition 3.5.3)
- Aho, A. V., 166, 233
- ALGOL, 14, 20, 38 (History 2.2.4)
- algorithm, 14
- $\alpha$ -equivalence ( $\equiv$ )
  - $\lambda$ -terms, 42 (Definition 2.3.2)
  - $\lambda a$ -terms, 101 (Definition 3.4.8)
  - $\lambda x$ -terms, 64 (Definition 3.1.2)
- alphabet, 52 (Definition 2.6.1)
- $\mathcal{A}\mathcal{M}\mathcal{S}\text{-}\mathcal{L}\mathcal{A}\mathcal{T}\mathcal{E}\mathcal{X}$ , 2, 210 (Example A.1.1)
- analysis, 203 (Code 6.9.7)
- antisymmetric, 32 (Definition 2.1.5)
- Anything, 166 (Code 6.3.1)
- App, 167 (Code 6.3.1)
- application, 41 (Definition 2.3.1), 54 (Notation 2.6.5)
- applicative TRS, 54 (Example 2.6.7)
- applying sequences, 37 (Notation 2.2.1)
- Ariola, Z. M., 4, 26, 27, 105 (History 3.4.13), 107 (Comparison 3.4.18), 110 (Comparison 3.5.7), 144 (Discussion 5.2.4), 146 (Comparison 5.2.8), 233, 234
- arity, 54 (Notation 2.6.5)
- arrows, 31 (Notation 2.1.4)
- ARS, *see* reduction
- artificial languages, 13
- Arvind, 26, 146 (Comparison 5.2.8), 233
- assert, 157 (Code 6.1.3)
- atoi, 169 (Code 6.3.2)
- Australian Research Council, 4
- AUTOMATH, 50
- axiom, 40 (Definition 2.2.6), 46

**B**

$\xrightarrow{B}$ , 86 (Definition 3.3.2)  
 $\xrightarrow{\bar{B}}$ , 65 (Definition 3.1.4)  
 back-pointer, 144  
 Backus, J., 19, 234  
 Backus-Naur form, *see* BNF  
 Barendregt, H. P., 4, 10 (Contribution 1.1.1),  
 22, 41, 58, 66 (Remark 3.1.7), 96,  
 146, 234  
 Benaïssa, Z.-E.-A., 86, 87, 112, 132 (Com-  
 parison 4.4.3), 234  
 Berners-Lee, T., 18, 234  
 $\xrightarrow{\beta}$ , 43 (Definition 2.3.6)  
 $\xrightarrow{\beta}$ -admissible, 102 (Definition 3.4.9)  
 $\beta$ -reduction, 43 (Definition 2.3.6)  
 $\#_{\beta_a}$ , 105 (Definition 3.4.14)  
 $\xrightarrow{\beta_h}$ , 47 (Definition 2.4.3)  
 $\xrightarrow{\beta_N}$ , 48 (Definition 2.4.5)  
 $\xrightarrow{\beta_{NF}}$ , 50 (Definition 2.5.1)  
 $\xrightarrow{\beta_V}$ , 48 (Definition 2.4.5)  
 $\xrightarrow{\beta_W}$ , 47 (Definition 2.4.3)  
 big-step semantics, 46 (History 2.4.2)  
 binary relation, 30 (Notation 2.1.1)  
 Bird, R., 15, 234  
 Bloo, R., 4, 10 (Contribution 1.1.1), 63, 83  
 (Remark 3.2.16), 112, 135 (Exam-  
 ple 5.1.1), 142, 153, 234, 235, 242  
 BNF, 38 (History 2.2.4)  
 bootstrap, 203 (Code 6.9.9)  
 bound variable naming, 42 (Convention 2.3.4)  
 bound variables (bv), 55 (Definition 2.6.9)  
 Briaud, D., 86, 87, 234  
 bug, 193 (Code 6.7.2)  
 $\xrightarrow{B_V}$ , 86 (Definition 3.3.2)  
 Burn, G. L., 112, 235  
 but, 157 (Code 6.1.4)  
 bv, 178 (Code 6.4.1)  
 $\xrightarrow{bx}$ , 65 (Definition 3.1.4)  
 $\xrightarrow{bx_{gc}}$ , 65 (Definition 3.1.4)  
 $\xrightarrow{bx_N}$ , 118 (Definition 4.2.1)  
 $\xrightarrow{bx_V}$ , 125 (Definition 4.3.1)

**C**

Cailliau, R., 18, 235  
 Call-by-Name, *see* CBN

Call-by-Name machines, 123 (Comparison 4.2.16)  
 Call-by-Need, 110 (Comparison 3.5.7)  
 Call-by-Text, 20  
 Call-by-Value, *see* CBV  
 Call-by-Value machines, 128 (Comparison 4.3.5)  
 CAM, 21  
 Cardelli, L., 11 (Contribution 1.1.1), 63, 233  
 Carnegie Mellon University, 4  
 Categorical Abstract Machine, 21  
 CBN, 20, 21, 45, 48 (Definition 2.4.5), 117  
 for  $\lambda x$ , 118 (Definition 4.2.1)  
 CBV, 45, 48 (Definition 2.4.5), 124  
 for  $\lambda x$ , 125 (Definition 4.3.1)  
 central processing unit, 19  
 Centrum der Wiskunde en Informatica, 4  
 Chalmers Tekniska Högskola/Göteborgs Uni-  
 versitet, 4  
 Check, 157 (Code 6.1.3)  
 check, 157 (Code 6.1.3)  
 check, 157 (Code 6.1.3)  
 checkCRS, 162 (Code 6.2.8)  
 checking, 157 (Code 6.1.3)  
 checkLHS, 162 (Code 6.2.7)  
 checkMterm, 161 (Code 6.2.5)  
 checkRHS, 162 (Code 6.2.7)  
 checkRule, 161 (Code 6.2.7)  
 Church, A., 9, 35, 41, 235  
 thesis, 20  
 Church-Rosser, *see* CR  
 Claus, V., 235, 236  
 closed, 52 (Definition 2.6.1), 64 (Definition  
 3.1.2), 160 (Code 6.2.5)  
 $\lambda$ -terms ( $\Lambda^\circ$ ), 42 (Definition 2.3.2)  
 $\lambda x$ -terms ( $\Lambda x$ ), 64 (Definition 3.1.2)  
 closed terms, 118  
 closure, 33 (Notation 2.1.8)  
 closure rules, 46 (History 2.4.2)  
 Collins, 13, 235  
 comb, 183 (Code 6.5.1)  
 combinator, 44 (Notation 2.3.8)  
 Combinatory Logic  
 as CRS, 54 (Example 2.6.7)  
 combinatory reduction system, *see* CRS  
 comparison between  $\lambda v$  and  $\lambda x$ , 84 (Discus-  
 sion 3.3.1)  
 compatible relation, 39

- compile, 116
  - complain, 159 (Code 6.1.7)
  - complement, 30 (Notation 2.1.1)
  - complete, 15, 35 (Definition 2.1.15)
  - complexity, 20, 111 (Discussion 3.5.8), 136
  - composition, 32 (Definition 2.1.6), 32 (Notation 2.1.7)
  - composition of substitution
    - breaks PSN, 75 (Remark 3.2.1)
  - compositional, 40 (Definition 2.2.6)
    - function, *see* derivor
  - computability, 20
  - computation steps, 16
  - computer, 19
  - concatenate sequences ( $\cdot$ ), 37 (Notation 2.2.1)
  - configuration, 19, 116
  - confluence, *see* CR
  - conservative extension, 34 (Definition 2.1.12)
    - $\lambda v$  of  $\lambda\beta$ , 89 (Theorem 3.3.8)
    - $\lambda xgc$  of  $\lambda\beta$ , 74 (Theorem 3.1.17)
    - $\lambda xgca$  of  $\lambda\beta$ , 109 (Corollary 3.5.6)
    - $\lambda xgca$  of  $\lambda xgc$ , 109 (Theorem 3.5.5)
  - construction, 52 (Definition 2.6.1)
  - construction ( $::=$ ), 38 (Notation 2.2.3)
  - context, 39 (Definition 2.2.5)
  - context-free, 39 (History 2.2.4)
  - contextual, 39 (Definition 2.2.5)
    - reduction, 39
  - contextual closure, 98
  - continuation-passing style, *see* CPS
  - contractor, 53 (Definition 2.6.1)
  - contractum, 57 (Definition 2.6.12)
  - converse, 31 (Definition 2.1.3)
  - converse, 192 (Code 6.6.7)
  - conversion, 32 (Definition 2.1.5)
  - copying, 103 (Definition 3.4.10), 111 (Discussion 3.5.8), 144
    - overhead, 108
  - correct, 15
  - Cousineau, G., 21, 235
  - CPS, 49 (Definition 2.4.8)
  - CR, 34 (Definition 2.1.13), 44 (Theorem 2.3.11)
    - $\lambda xjgc$ , 78 (Theorem 3.2.4)
    - $\lambda xgc$ , 74 (Corollary 3.1.18)
    - $xgc$ , 69 (Proposition 3.1.10)
  - Crégut, P., 112, 123 (Comparison 4.2.16), 236
  - critical pairs, 191 (Code 6.6.5)
  - criticalpairs, 191 (Code 6.6.5)
  - CRS, 51, 53 (Definition 2.6.1), 162 (Code 6.2.8)
    - abbreviations, 54 (Notation 2.6.5)
    - datatype, 159
    - explicification, 137 (Definition 5.1.9), 192
    - inference rule, 197
    - input, 201 (Code 6.9.3)
    - output, 202 (Code 6.9.5)
    - parser, 166 (Code 6.3.1)
    - restricted, 53 (Notation 2.6.2)
    - rewrite rules, 52 (Definition 2.6.1)
    - terms, 52 (Definition 2.6.1)
    - tokens, 171 (Code 6.3.3)
  - CRS, 162 (Code 6.2.8), 166 (Code 6.3.1)
  - Crs, 168 (Code 6.3.1)
  - CRSa, 143 (Definition 5.2.1)
  - CRSar, 145 (Definition 5.2.5)
  - Curien, P.-L., 11 (Contribution 1.1.1), 21, 63, 233, 235
  - current working CRS, 201 (Code 6.9.2)
  - Curry, H. B., 10 (Contribution 1.1.1), 46 (Definition 2.4.1), 236
    - fixed point combinator ( $Y$ ), 44 (Notation 2.3.8)
  - cyclic, 145 (Definition 5.2.6)
  - cyclic  $\lambda$ -graph reduction, 145 (Example 5.2.7), 146 (Comparison 5.2.8)
  - cyclic addressing
    - CRS, 145 (Definition 5.2.5)
  - cyclic sharing
    - terms ( $\_ wfar$ ), 145 (Definition 5.2.6)
  - cyclic sharing CRS, 145 (Definition 5.2.5)
  - cyclic substitutions, 146 (Comparison 5.2.8)
- ## D
- DACTL, 197
  - Danthine, A. A. S., 197, 236
  - Danvy, O., 4, 49, 236
  - DART, 4
  - de Bakker, J. W., 22, 58, 96, 146, 234
  - de Bruijn, N. G., 10 (Contribution 1.1.1), 29, 61, 96 (Discussion 3.3.22), 235

index, 84  
 level, 93  
 de Vries, F. J., 144 (Discussion 5.2.4), 238  
 de Vrijer, R. C., 50, 235, 240  
 definable extensions, 51  
**delay**, 123 (Comparison 4.2.16)  
 $\Delta(-)$ , *see* unravel  
 derivor, 40  
 Det Danske Forskningsråd, 4  
 deterministic, 31 (Definition 2.1.3)  
 Dezani, M., 7, 10 (Contribution 1.1.1), 62,  
 215 (Example A.2.3), 239, 253  
 diagrammatic assertion, 31 (Notation 2.1.4)  
 diagrammatic induction, 41 (Notation 2.2.8)  
 diagrammatic propositions, 31 (Notation 2.1.4)  
 diagrammatic reasoning, 31 (Notation 2.1.4)  
 dialogue, 207 (Example A.1.1)  
 diamond property ( $\diamond$ ), 34 (Definition 2.1.13)  
 DIKU, 1, 3  
 directly abstractable, 105 (History 3.4.13)  
 documentation, 200 (Code 6.9.1)  
 domain, 30 (Notation 2.1.1)  
 drop from sequence ( $/$ ), 37 (Notation 2.2.1)  
 dummy address ( $\epsilon$ ), 129 (Definition 4.4.1)  
 duplication, 108  
**E**  
 Edwards, D. J., 26, 45, 239  
 Ehrig, H., 27, 235, 236  
 either, 157 (Code 6.1.3)  
 empty map, 37 (Notation 2.2.1)  
 empty sequence ( $\epsilon$ ), 36 (Notation 2.2.1)  
 environment frames, 132 (Comparison 4.4.3)  
 environment-based evaluation, 117  
 $\epsilon$   
   address, 129 (Definition 4.4.1)  
   sequence, 36 (Notation 2.2.1)  
 Eq, 161 (Code 6.2.6)  
 $\mathbb{EQ}$ , 20  
 equality ( $=$ ), 30  
 $\equiv$ , *see*  $\alpha$ -equivalence  
 $\equiv$ , 161 (Code 6.2.6), 178 (Code 6.4.2)  
 equivalence, 32 (Definition 2.1.5)  
 equivalence closure, 33 (Notation 2.1.8)  
 ESCRS, 135 (Definition 5.1.2)  
 ESCRSa, 144 (Definition 5.2.3)

$\eta$ -reduction, 44 (Remark 2.3.9)  
 evaluation strategies, 44  
 examples of properties as diagrams, 31 (Def-  
 inition 2.1.3)  
 exhaustive reduction, 185 (Code 6.5.4)  
 explicification, 193 (Code 6.7.1)  
 explicify, 193 (Code 6.7.1)  
 explicit garbage collection, 63  
 explicit metaapplications., 135 (Definition 5.1.2)  
 explicit naming, 90 (Definition 3.3.11)  
 explicit recursion, 144 (Discussion 5.2.4)  
 explicit substitution, 65 (Definition 3.1.4)  
 explicit substitution and sharing CRS, 144  
 (Definition 5.2.3)  
 explicit substitution CRS, *see* ESCRS  
 extended BNF, 39 (History 2.2.4)  
 extensionality, 44 (Remark 2.3.9)  
 external position, 90 (Comparison 3.3.10)

**F**

fair complexity measure, 137 (Discussion 5.1.8)  
 Fairbairn, J., 131 (Comparison 4.4.3), 236  
 Fasel, H., 15, 237  
 fastfib, 25 (Code 1.3.3)  
 Felleisen, M., 106 (Comparison 3.4.18), 110  
 (Comparison 3.5.7), 144, 233, 236  
 Feys, R., 10 (Contribution 1.1.1), 46 (Defi-  
 nition 2.4.1), 236  
 fib, 24 (Code 1.3.2)  
 Fibonacci number, 22 (Definition 1.3.1), 24  
 (Code 1.3.2), 212 (Code A.1.3)  
   fast, 25 (Code 1.3.3)  
 Fielding, R., 18, 234  
 final restriction ( $\rightarrow$ ), 35 (Definition 2.1.15)  
 finite sequence, 36 (Notation 2.2.1)  
 first order functional programming, 146  
 first order rewriting systems, 53 (Remark  
 2.6.3)  
 fixed point combinator, 44 (Notation 2.3.8)  
 flat, 40 (Definition 2.2.6)  
**force**, 123 (Comparison 4.2.16)  
 free addresses, 142 (Definition 5.2.1)  
 free variable, *see* fv  
 free variable matching, 137 (Remark 5.1.7)  
 free variable matching constraint, 159 (Code  
 6.2.1), 174, 176 (Code 6.4.1)



free variable pattern, 137 (Remark 5.1.7)  
 fresh, 101, 174  
 fresh, 160 (Code 6.2.2)  
 fresh address, 142 (Definition 5.2.1)  
 Friedman, D. P., 106 (Comparison 3.4.18),  
 144, 236  
 Frystyk, H., 18, 234  
 full laziness, 105 (History 3.4.13)  
 fully abstract, 15  
 function, 31 (Definition 2.1.3)  
   from sequence, 37 (Notation 2.2.1)  
 function definition, 14  
 function symbols, 52 (Definition 2.6.1)  
   CRS, 159 (Code 6.2.1)  
 functional, 14  
 functional programming, 17, 23  
 functional programming languages, 14  
 fv  
   contexts, 42 (Definition 2.3.2)  
   CRS, 52 (Definition 2.6.1)  
    $\lambda$ -terms, 42 (Definition 2.3.2)  
    $\lambda a$ -terms, 100 (Definition 3.4.8)  
    $\lambda x$ -terms, 64 (Definition 3.1.2)  
 fv, 160 (Code 6.2.5)

**G**

G-machine, 26, 131 (Comparison 4.4.3)  
 Gabbay, D. M., 30, 238  
 Galois connection, 33 (Remark 2.1.11)  
 $\gamma$ -node, 103 (History 3.4.12)  
 Ganzinger, H., 142, 235  
 garbage, 65 (Definition 3.1.4)  
 garbage collection, 65 (Definition 3.1.4), 72  
   (Remark 3.1.14), 119 (Remark 4.2.3)  
 garbage-free, 66 (Notation 3.1.6)  
 $\xrightarrow{\text{gc}}$ , 65 (Definition 3.1.4)  
 general replacement system, *see* reduction  
 Geuvers, J. H., 50, 83 (Remark 3.2.16), 234,  
 235, 240  
 Glauert, J. R. W., 22, 26, 58, 96, 112, 146,  
 197, 234, 236  
 Goal, 166 (Code 6.3.1)  
 Gödel, K., 20, 236  
 Gordon, A. D., 96 (Discussion 3.3.22), 236  
 grammar, 13, 39 (History 2.2.4)  
 graph bisimilarity, 105 (History 3.4.13)

graph matching ordering, 105 (History 3.4.13)  
 graph reducible, 99 (Comparison 3.4.7)  
 graph reduction, 26  
 Graph rewrite systems, 146 (Comparison 5.2.8)  
 GRS, 146 (Comparison 5.2.8)  
 Grue, K., 112, 236

**H**

Hannan, J., 114, 124 (Comparison 4.2.16),  
 128 (Comparison 4.3.5), 236  
 Hardin, T., 36  
 Harper, R., 20, 45, 239  
 Hart, T. P., 26, 45, 239  
 hastrans, 158 (Code 6.1.5)  
 head reduction, 47 (Definition 2.4.3)  
 headaches, 144 (Discussion 5.2.4)  
 heap, 131 (Comparison 4.4.3)  
 Henderson, P., 111 (Discussion 3.5.8), 114,  
 123 (Comparison 4.2.16), 128 (Com-  
 parison 4.3.5), 237  
 Hermenegildo, M., 10 (Contribution 1.1.1),  
 63, 91, 237  
 high-level, 14  
 higher order rewriting, 51  
 hole ( $\square$ ), 39 (Definition 2.2.5)  
 Hsiang, J., 11 (Contribution 1.1.1), 84, 93,  
 239  
 HTTP, 18  
 Hudak, P., 15, 155, 237  
 Huet, G., 30, 237  
 Hughes, J. M., 4, 105 (History 3.4.13), 237  
 HUGS, 199, 203 (Code 6.9.9), 207 (Example  
 A.1.1), 220 (Example A.3.2)

**I**

I, 44 (Notation 2.3.8)  
 i/o errors, 159 (Code 6.1.7)  
 identity  
   sequence ( $\iota$ ), 37 (Notation 2.2.1)  
 identity map ( $\iota$ ), 37 (Notation 2.2.1)  
 implementing, 14  
 implicit renaming, 66 (Remark 3.1.7)  
 indirection node, 98 (Remark 3.4.3)  
 inductive structures, 38 (Notation 2.2.3)  
 inference metavariables, 40 (Definition 2.2.6)

inference rules, 40 (Definition 2.2.6), 46, 197  
 (Code 6.8.1)  
 infinite lists, 25 (Code 1.3.4)  
 infinite sequence, 37 (Notation 2.2.1)  
 innermost, 183 (Code 6.5.2)  
 input, 165  
 inside garbage, 90 (Comparison 3.3.10)  
 instruction, 116  
 instruction cycle, 19  
 interactive, 199  
 interactive operation, 201 (Code 6.9.2)  
 internal position, 90 (Comparison 3.3.10)  
 inversetrans, 158 (Code 6.1.5)  
 $\iota$ , 37 (Notation 2.2.1)  
 istrans, 158 (Code 6.1.5)  
 item, 91

**J**

Jeffrey, A., 22, 112, 237  
 Johnsson, T., 4, 26, 105 (History 3.4.13), 131  
 (Comparison 4.4.3), 237  
 Jones, M. P., 199, 237  
 Jones, N., 3  
 Junk, 166 (Code 6.3.1), 168 (Code 6.3.1)

**K**

K, 44 (Notation 2.3.8)  
 Kahn, G., 21, 40 (Remark 2.2.7), 131 (Comparison 4.4.3), 236, 237  
 Kahrs, S., 155, 237  
 Kamareddine, F., 10 (Contribution 1.1.1),  
 63, 75 (Remark 3.2.1), 91, 237  
 Katholieke Universiteit Nijmegen, 4  
 Kennaway, J. R., 22, 26, 58, 77, 96, 112, 144  
 (Discussion 5.2.4), 146, 197, 234,  
 236, 238  
 Kleene, S. C., 20, 238  
 Kleene's star (\*), 38 (Notation 2.2.3)  
 Klop, J. W., 3, 12 (Contribution 1.1.3), 27,  
 29, 30, 51, 57 (Theorem 2.6.15),  
 105 (History 3.4.13), 107 (Comparison  
 3.4.18), 133, 144 (Discussion  
 5.2.4), 146 (Comparison 5.2.8), 174,  
 233, 234, 238  
 Knuth, D. E., 2, 22, 58, 163, 238  
 Københavns Universitet, 3

Koopman, P. W. M., 26, 112, 238  
 Krieg-Brückner, B., 132 (Comparison 4.4.3),  
 152, 241  
 Krivine, J.-J.  
 machine, 123 (Comparison 4.2.16)  
 $K^*$ , 44 (Notation 2.3.8)

**L**

LABEL, 26, 144 (Discussion 5.2.4)  
 $\Lambda$ , 42 (Definition 2.3.2)  
 $\lambda$ -calculus, 15  
 as CRS, 53 (Example 2.6.4)  
 readable CRS, 54 (Example 2.6.6)  
 $\lambda$ -definability, 20  
 $\lambda$ -graph, 103 (History 3.4.12)  
 $\Lambda$ -inferences, 45 (Definition 2.4.1)  
 $\lambda$ -preterms, 41 (Definition 2.3.1)  
 $\lambda$ -space, 61  
 $\lambda$ -terms ( $\Lambda$ ), 42 (Definition 2.3.2)  
 $\lambda$ a-preterms, 100 (Definition 3.4.8)  
 $\lambda$ a-reduction, 105 (Definition 3.4.14)  
 $\lambda$ a-terms ( $\Lambda$ a), 101 (Definition 3.4.8)  
 $\lambda\chi$ , 93 (Definition 3.3.18)  
 $\lambda\chi$ -terms ( $\Lambda\chi$ ), 93 (Definition 3.3.18)  
 $\lambda\ell$ , 49 (History 2.4.10)  
 $\lambda_{\text{let}}$ , 110 (Comparison 3.5.7)  
 $\lambda_{\mathbb{N}}$ , 48 (History 2.4.7)  
 $\lambda\text{NF}$ , 50 (Definition 2.5.1)  
 $\Lambda^\circ$ , 42 (Definition 2.3.2)  
 $\lambda$ s, 91 (Definition 3.3.14)  
 $\lambda$ s-reduction, 91 (Definition 3.3.14)  
 $\lambda$ s-terms ( $\Lambda$ s), 91 (Definition 3.3.14)  
 $\lambda\sigma$ , 93 (Definition 3.3.21)  
 $\lambda\sigma$ -terms ( $\Lambda\sigma$ ), 93 (Definition 3.3.21)  
 $\lambda\nu$ , 86 (Definition 3.3.2)  
 $\lambda_{\mathbb{V}}$ , 48 (History 2.4.7)  
 $\Lambda x$ , 64 (Definition 3.1.2)  
 $\lambda x$ , 66  
 $\lambda x$ -preterms, 63 (Definition 3.1.1)  
 $\lambda x$ -terms, 64 (Definition 3.1.2)  
 $\lambda x$ -terms ( $\Lambda x$ ), 64 (Definition 3.1.2)  
 $\lambda xa$ -terms, 107 (Definition 3.5.1)  
 $\lambda xa_{\mathbb{N}}$   
 reduction step, 129 (Definition 4.4.1)  
 $\lambda xgc$ -reduction, 65 (Definition 3.1.4)  
 as CRS, 134 (Example 5.1.1)

- $\lambda x_{gc}$ 
    - reduction step, 108 (Definition 3.5.2)
  - $\lambda x_N M$ , 122
    - correctness, 122 (Theorem 4.2.13)
  - $\lambda x_V M$ , 128
  - $\lambda x_N$ -machine, 121 (Definition 4.2.10)
  - $\lambda x_N$ , 118 (Definition 4.2.1)
    - correctness, 118 (Lemma 4.2.2)
  - $\Lambda x^\circ$ , 64 (Definition 3.1.2)
  - $\lambda x_V$ , 125 (Definition 4.3.1)
  - Lampert, L., 2, 163, 210 (Example A.1.1), 238
  - Landin, P. J., 11 (Contribution 1.1.2), 113, 238
  - language, 13
  - $\mathbb{A}T_{EX}$ , 2, 210 (Example A.1.1)
  - Launchbury, J., 22, 40 (Remark 2.2.7), 107 (Comparison 3.4.18), 132 (Comparison 4.4.3), 239
  - lazy, 49 (History 2.4.10)
  - lazy machines, 131 (Comparison 4.4.3)
  - LC, 34 (Definition 2.1.13)
  - left hand side, *see* LHS
  - left-linear, 57 (Definition 2.6.14), 186 (Code 6.6.1)
  - leftlinear, 186 (Code 6.6.1)
  - leftmost, 183 (Code 6.5.2)
  - length induction, 36
  - length of sequence ( $\#$ ), 36 (Notation 2.2.1)
  - Lescanne, P., 4, 10, 11 (Contribution 1.1.1), 62, 84, 86, 87, 93, 112, 115, 132 (Comparison 4.4.3), 234, 239
  - $\leq$ , 178 (Code 6.4.2)
  - Lester, D., 131 (Comparison 4.4.3), 240
  - Levin, M., 26, 45, 239
  - Lévy, J.-J., 11 (Contribution 1.1.1), 63, 96, 233, 239
  - LHS, 52 (Definition 2.6.1)
  - linear, 186 (Code 6.6.1)
  - LISP, 20, 21, 45, 144 (Discussion 5.2.4)
  - literate Haskell, 156 (Notation 6.1.1)
  - local, 115 (Definition 4.1.1)
  - local confluence, *see* LC, 192 (Code 6.6.8), 230 (Code A.4.5)
  - locality, 74 (Discussion 3.1.19), 111 (Discussion 3.5.8)
  - locally admissible, 109 (Definition 3.5.3)
  - locally R-admissible, 109 (Definition 3.5.3)
  - locallyconfluent, 192 (Code 6.6.8)
  - location equality, 20
  - logic, 44 (Remark 2.3.9)
  - low-level, 14
- ## M
- Mabs, 160 (Code 6.2.3)
  - machine
    - abstract, 115 (Definition 4.1.1)
  - machine state, 128 (Comparison 4.3.5)
  - Macquarie University, 4
  - Maibaum, T. S. E., 30, 238
  - many-to-many relation, 33 (Remark 2.1.11)
  - many-to-one relation, 33 (Remark 2.1.11)
  - map, 31 (Definition 2.1.3)
    - from sequence, 37 (Notation 2.2.1)
  - Mapp, 160 (Code 6.2.3)
  - Maraist, J., 110 (Comparison 3.5.7), 233
  - Maranget, L., 4
  - match
    - for CRS, 56 (Definition 2.6.12)
  - match, 175 (Code 6.4.1)
  - matches, 55 (Definition 2.6.8)
  - MatchFail, 175 (Code 6.4.1)
  - matching, 175 (Code 6.4.1)
  - Mauny, M., 21, 235
  - maximal free subexpressions, 105 (History 3.4.13)
  - McCarthy, J., 17, 26, 45, 239
  - Mcon, 160 (Code 6.2.3)
  - mechanical evaluation, 113
  - mechanically computable, 15
  - Melliès, P.-A., 7, 10 (Contribution 1.1.1), 62, 215 (Example A.2.3), 239, 253
  - metavariables (mv), 52 (Definition 2.6.1)
  - meta ... , 57 (Definition 2.6.12)
  - metaabstraction, 52 (Definition 2.6.1), 54 (Notation 2.6.5)
  - metaapplication, 52 (Definition 2.6.1)
  - metacontractum, 57 (Definition 2.6.12)
  - metaredex, 57 (Definition 2.6.12)
  - metareduction, 57 (Definition 2.6.13)
  - metarewrite, 57 (Definition 2.6.12)
  - MetaTerm, 167, 168 (Code 6.3.1)

- metaterm, 57 (Definition 2.6.12)
  - metaterm equality, 161 (Code 6.2.6)
  - metaterm input, 202 (Code 6.9.4)
  - MetaTermList, 167 (Code 6.3.1)
  - MetaVar, 168 (Code 6.3.1)
  - metavariables
    - CRS, 160 (Code 6.2.4)
  - Miller, D., 114, 124 (Comparison 4.2.16), 128 (Comparison 4.3.5), 236
  - Milner, R., 20, 45, 239
  - minimal derivations, 90 (Comparison 3.3.10)
  - minimalise, 191 (Code 6.6.6)
  - minimalise CRS, 191 (Code 6.6.6)
  - mkAbs, 170 (Code 6.3.2)
  - mkApp, 169 (Code 6.3.2)
  - mktrans, 158 (Code 6.1.5)
  - mkTuple, 169 (Code 6.3.2)
  - mkVect, 170 (Code 6.3.2)
  - model, 15
  - Mogensen, T., 4, 166, 239
  - Moore, R. R., 2, 4, 242
  - Mosses, P., 117, 239
  - motivations, 17
  - Mterm, 160 (Code 6.2.3)
  - $(\mu)$ , 46 (Definition 2.4.1)
  - $(\mu_N)$ , 48 (Definition 2.4.5)
  - $(\mu_V)$ , 48 (Definition 2.4.5)
  - Mv, 160 (Code 6.2.3)
  - mv, 52 (Definition 2.6.1), 160 (Code 6.2.4)
  - mv, 160 (Code 6.2.4)
  - Mvar, 160 (Code 6.2.3)
- N**
- $\xrightarrow{N}$ , 48 (Definition 2.4.5)
  - n-ary relation, 30 (Notation 2.1.1)
  - namefree, 50
  - natural
    - sequence, 37 (Notation 2.2.1)
  - natural languages, 13
  - natural semantics, 40 (Remark 2.2.7)
  - Naur, P., 14, 38 (History 2.2.4), 240
  - Nederpelt, R. P., 4, 50, 75 (Remark 3.2.1), 235, 237, 240
  - $\xrightarrow{need}$ , 110
  - Newman, M. H. A., 41 (Lemma 2.2.9), 240
  - nf, 35 (Definition 2.1.15)
  - nf, 184 (Code 6.5.3)
  - n-fold composition ( $\xrightarrow{n}$ ), 32 (Notation 2.1.7)
  - nfs, 185 (Code 6.5.4)
  - Nielson, H. R., 21, 112, 241
  - Nijman, A. J., 22, 58, 96, 146, 234
  - Nipkow, T., 51, 240
  - noetherian, *see* SN
  - non-standard reductions, 153
  - nonoverlapping, 57 (Definition 2.6.14), 186 (Code 6.6.2)
  - nonoverlapping, 186 (Code 6.6.2)
  - Nordisk Forskerakademi, 4
  - normal forms, *see* nf
    - $\lambda xgc$ , 66 (Notation 3.1.6)
  - normalisation, 35 (Definition 2.1.15)
  - notrans, 158 (Code 6.1.5)
  - $(\nu)$ , 46 (Definition 2.4.1)
  - number of symbols, 38 (Notation 2.2.3)
- O**
- observe, 15
  - occurrence, 56 (Definition 2.6.12)
  - Odersky, M., 110 (Comparison 3.5.7), 233
  - $\Omega$ , 44 (Notation 2.3.8)
  - $\omega$ , 44 (Notation 2.3.8)
  - onestep, 184 (Code 6.5.3)
  - openfile, 204 (Code 6.9.9)
  - operation, 16
  - operational, 16
  - oracle, 180 (Code 6.4.4)
  - Oregon Graduate Institute, 4
  - orthogonal, 57 (Definition 2.6.14), 191 (Code 6.6.4)
  - orthogonal, 191 (Code 6.6.4)
  - others, 14, 38 (History 2.2.4), 155, 237, 240
  - outermost, 183 (Code 6.5.2)
  - Overlap, 186 (Code 6.6.2)
  - overlaps, 230 (Code A.4.4)
  - overlaps, 187 (Code 6.6.2)
  - overlay, 188 (Code 6.6.3)
- P**
- parallel reduction, 57, 96, *see* sharing extension, 99
  - parallel substitution, 75 (Remark 3.2.1)
  - ParseTree, 168 (Code 6.3.1)

- partial, 31 (Definition 2.1.3)
  - partial order, 32 (Definition 2.1.5)
  - PASCAL, 20
  - Path, 179 (Code 6.4.2)
  - pattern, 52 (Definition 2.6.1)
  - PCF, 217
    - with explicit sharing, 220 (Example A.3.2)
    - with explicit sharing and substitution, 225 (Code A.3.3)
  - PCF+pairs, 217 (Example A.3.1)
  - Petri nets, 197
  - Peyton Jones, S. L., 4, 26, 105 (History 3.4.13), 112, 129, 131 (Comparison 4.4.3), 155, 235, 237, 240
  - Pfenning, F., 49, 236
  - Pisano, L., 22, 240
  - Plasmeijer, M. J., 4, 22, 27, 58, 96, 105 (History 3.4.13), 112, 131 (Comparison 4.4.3), 146, 234, 238, 240–242
  - Plotkin, G. D., 11 (Contribution 1.1.2), 16, 47, 112, 114, 117, 133, 217, 241
    - PCF, 217 (Example A.3.1)
  - pointers, 20
  - Precious, 166 (Code 6.3.1)
  - premv, 170 (Code 6.3.2)
  - preop, 170 (Code 6.3.2)
  - preserve strong normalisation, *see* PSN
  - preserve weak normalisation, *see* PWN
  - presym, 170 (Code 6.3.2)
  - preterms, 38, 52 (Definition 2.6.1)
    - CRS, 160 (Code 6.2.3)
    - $\lambda$ , 41 (Definition 2.3.1)
    - $\lambda a$ , 100 (Definition 3.4.8)
    - $\lambda x$  &  $\lambda xgc$ , 63 (Definition 3.1.1)
  - pretty interaction, 229 (Code A.4.3)
  - prettyprinting, 211 (Example A.1.2)
  - primitive, 115 (Definition 4.1.1)
  - print reduction, 228 (Code A.4.2)
  - printing
    - CRS, 163, 165 (Code 6.2.12)
    - CRS rule, 165 (Code 6.2.11)
    - function symbols, 163 (Code 6.2.9)
    - metaterms, 163 (Code 6.2.10)
  - production, 38 (Notation 2.2.3)
  - program analysis, 152
  - program transformations, 25, 153
  - programming, 14
  - programming language, 14
    - functional, 14
  - projection, 33 (Definition 2.1.10)
    - $\lambda xgc$ , 73 (Lemma 3.1.16)
  - PROLOG, 21
  - PSN, 35 (Definition 2.1.15)
    - R $x$  of R, 142 (Theorem 5.1.19)
    - lambdaxgca
      - $\lambda xgca$ , 109 (Corollary 3.5.6)
    - counterexample of Melliès, 76 (Remark 3.2.1)
    - for  $\lambda xgc$ , 82 (Corollary 3.2.13)
    - garbage-free ( $\lambda x\downarrow gc$ ), 78 (Theorem 3.2.6)
    - $\lambda x$ , 93 (Corollary 3.3.20)
    - $\lambda s$ , 93 (Corollary 3.3.17)
    - $\lambda v$ , 89 (Theorem 3.3.9)
    - strict explicit naming, 91 (Theorem 3.3.13)
  - pure  $\lambda xgc$ -term, 66 (Notation 3.1.6)
  - pure functional programming languages, 17
  - Purushothaman, S., 132 (Comparison 4.4.3), 152, 241
  - PWN, 35 (Definition 2.1.15)
- Q**
- qsymbols, 210 (Example A.1.1)
- R**
- R-admissible, 102 (Definition 3.4.9), 104 (History 3.4.13)
  - R-contraction, 105 (Definition 3.4.14)
  - R-copies, 103 (Definition 3.4.10)
  - rand node, 103 (History 3.4.12)
  - range, 30 (Notation 2.1.1)
  - Raoult, J. C., 27, 241
  - RATATOSK, 166
  - rator node, 103 (History 3.4.12)
  - readMterm, 169 (Code 6.3.1)
  - readParseTree, 169 (Code 6.3.1)
  - rebind, 178 (Code 6.4.1)
  - recursive equations, 15, 17, 146
  - recursive functions, 15, 20
  - Redex, 178 (Code 6.4.2)
  - redex, 56 (Definition 2.6.12), 178 (Code 6.4.2)
  - redexes, 179 (Code 6.4.2)
  - redexprefix, 183 (Code 6.5.2)

- reducible, 56 (Definition 2.6.12)
  - reduction, 15, 30 (Notation 2.1.1), 202 (Code 6.9.6)
    - $\beta$ , 43 (Definition 2.3.6)
    - $\beta_{NF}$ , 50 (Definition 2.5.1)
    - compositional, 40
    - CRS, 57 (Definition 2.6.13)
    - flat, 40
    - for CRS, 57 (Definition 2.6.13)
    - garbage-free ( $\lambda x \downarrow gc$ ), 77 (Definition 3.2.2)
    - $\lambda s$ , 93 (Definition 3.3.18)
    - $\lambda_{\text{let}}$ , 110 (Comparison 3.5.7)
    - $\lambda s$ , 91 (Definition 3.3.14)
    - $\lambda \sigma$ , 93 (Definition 3.3.21)
    - $\lambda v$ , 86 (Definition 3.3.2)
    - $\lambda xgc$  &  $\lambda x$ , 65 (Definition 3.1.4)
  - reduction graph, 43 (Example 2.3.7)
  - reduction in context, 46
  - reduction strategies
    - standard, 183 (Code 6.5.2)
  - reduction strategy, 183 (Code 6.5.1)
  - reduction systems, 16
  - references, 20
  - reflexive, 32 (Definition 2.1.5)
  - regular, 57 (Definition 2.6.14)
  - relations, 30 (Notation 2.1.1)
  - Rémy, J.-L., 63, 146 (Comparison 5.2.8), 241
  - rename, 161 (Code 6.2.6)
  - renaming ( $[-x := y]$ ), 42 (Definition 2.3.2)
    - $\lambda a$ -terms, 101 (Definition 3.4.8)
    - $\lambda x$ -terms, 64 (Definition 3.1.2)
  - representation, 140 (Lemma 5.1.14)
    - $\lambda xgc$ , 70 (Lemma 3.1.11)
  - representative, 38
  - restricted CRS, 53 (Notation 2.6.2)
  - restriction, 33 (Notation 2.1.9)
  - restriction ( $\cdot|_-$ ), 30 (Notation 2.1.1)
  - reversible, 192 (Code 6.6.7)
  - reversible CRS, 192 (Code 6.6.7)
  - Revesz, G., 67 (Comparison 3.1.9), 241
  - rewrite, 57 (Definition 2.6.12)
  - rewrite, 180 (Code 6.4.4)
  - rewrite rules, 52 (Definition 2.6.1)
    - CRS, 161 (Code 6.2.7)
  - rewrite step, 180 (Code 6.4.4)
  - rewriting, 24
  - Reynolds, J., 4
  - RHS, 53 (Definition 2.6.1)
  - right hand side, *see* RHS
  - rightmost, 183 (Code 6.5.2)
  - Ríos, A., 10 (Contribution 1.1.1), 63, 91, 237
  - Robson, J. D., 112, 235
  - rootname, 170 (Code 6.3.2)
  - Rose, E., 4, 21, 112, 241
  - Rose, K. H., 1, 2, 4, 56 (Remark 2.6.11), 63, 105 (History 3.4.13), 135 (Example 5.1.1), 142, 146 (Comparison 5.2.8), 163, 210 (Example A.1.1), 234, 235, 241, 242
  - Rosen, B. K., 30, 242
  - Rosser, J. B., 35, 235
  - Rouyer-Degli, J., 11 (Contribution 1.1.1), 84, 86, 87, 93, 234, 239
  - Rozenberg, G., 235, 236
  - RPLAC, 21, 26
  - Rule, 161 (Code 6.2.7), 167 (Code 6.3.1)
  - Rules, 167 (Code 6.3.1)
  - Rusinowitch, M., 63, 146 (Comparison 5.2.8), 241
  - rxMterm, 226 (Code A.4.1)
- ## S
- S, 44 (Notation 2.3.8)
    - de Bruijn's, 51 (History 2.5.3)
  - $\xrightarrow{s}$ , 91 (Definition 3.3.14)
  - safe for the valuation, 56 (Definition 2.6.9)
  - safe with respect to itself, 56 (Definition 2.6.9)
  - safeness of CRS, 55 (Definition 2.6.9)
  - Salkild, J., 131 (Comparison 4.4.3), 240
  - sample output, 230 (Code A.4.6)
  - satmv, 170 (Code 6.3.2)
  - satmv3, 171 (Code 6.3.2)
  - saturated, 137 (Definition 5.1.5), 159 (Code 6.2.1), 176 (Code 6.4.1)
  - saturated, 197 (Code 6.7.6)
  - scientific contributions, 3
  - Seaman, J., 132 (Comparison 4.4.3), 152, 241
  - SECD, 114
  - self-application combinator, 44 (Notation 2.3.8)
  - semantics, 13
  - semicompositional, 40 (Definition 2.2.6)

- Seq, 167 (Code 6.3.1)
- sequence, 36 (Notation 2.2.1)
- sequence algebra, 37 (Proposition 2.2.2)
- Sestoft, P., 3, 112, 132 (Comparison 4.4.3), 242
- Sethi, R., 166, 233
- sets, 157 (Code 6.1.4)
- shared, 97 (Definition 3.4.2), 145 (Definition 5.2.6)
- shared  $\beta$ -reduction, 105 (Definition 3.4.14)
- sharing, 20, 25, 96, 174
  - and de Bruijn indices, 111 (Discussion 3.5.9)
  - closure, 99 (Definition 3.4.5)
  - cyclic, 145
  - extension ( $\rightsquigarrow$ ), 98 (Definition 3.4.5)
  - graph reduction intuition, 98 (Remark 3.4.3)
  - reduction, 98 (Definition 3.4.5)
  - stable, 106 (Definition 3.4.16), 106 (Proposition 3.4.17)
  - stable under updating, 98 (Proposition 3.4.4)
  - syntax, *see* address
  - terms ( $\_wfa$ ), 97 (Definition 3.4.2)
- sharing introduction, 108
- Sheeran, M., 4
- showCRS, 165 (Code 6.2.12)
- showMterm, 163 (Code 6.2.10)
- showRule, 165 (Code 6.2.11)
- showSym, 163 (Code 6.2.9)
- Simple, 167 (Code 6.3.1)
- simple sharing CRS, 142 (Definition 5.2.1)
- simplification, 24
- size ( $|\_|$ ), 38 (Notation 2.2.3)
- Sleep, M. R., 22, 26, 58, 77, 96, 105 (History 3.4.13), 112, 144 (Discussion 5.2.4), 146, 197, 234, 236, 238, 240, 241
- small-step semantics, 46 (History 2.4.2)
- Smetsers, S., 112, 238, 242
- SML, 20, 45
- SN, 35 (Definition 2.1.15), 186 (Code 6.5.4)
  - $\xrightarrow{e}$  for ESCRS, 139
  - xgc, 69 (Proposition 3.1.10)
- sort, 158 (Code 6.1.6)
- sorting, 158 (Code 6.1.6)
- sound, 15
- space faithful, 115 (Definition 4.1.1)
- space local, 115 (Definition 4.1.1)
- Spiegel, M. R., 58 (Example 2.6.16), 242
- standard combinators, 44 (Notation 2.3.8)
- standard reductions, 153
- standardisation, 48
- Staples, J., 112, 242
- $*$ , 37 (Notation 2.2.1), 38 (Notation 2.2.3)
- state, 116
- stateless, 18
- Statens Naturvidenskabelige Forskningsråd, 4
- Steele, Jr., G. L., 48, 242
- stencil diagram, 30
- STG-machine, 131 (Comparison 4.4.3)
- store, 21, 131 (Comparison 4.4.3)
- Strategy, 183 (Code 6.5.1)
- strict explicit naming, 90 (Definition 3.3.12)
- strict subterm ordering ( $\triangleright$ ), 39 (Definition 2.2.5)
- strong, 184 (Code 6.5.2)
- strongly normalising, *see* SN
- structural, 46
- structural induction, 36, 38
- structure preserving, 142 (Definition 5.1.18)
- subreduction, 34 (Definition 2.1.12)
- Subst, 179 (Code 6.4.3)
- subst, 180 (Code 6.4.3)
- substitute, 55 (Definition 2.6.8)
- substitution, 43 (Definition 2.3.6), 55 (Definition 2.6.8), 111 (Discussion 3.5.8), 179 (Code 6.4.3)
  - CRS, 55 (Definition 2.6.8)
  - primitive, 116 (Proposition 4.1.2)
- substitution distribution, 195 (Code 6.7.3)
- substitution elimination, 195 (Code 6.7.4)
- substitution generation, 65 (Definition 3.1.4)
- substitution introduction, 193 (Code 6.7.2)
- substitution lemma, 44 (Proposition 2.3.10), 71
  - $\lambda xgc$ , 71 (Corollary 3.1.12)
- substitution-based evaluation, 117
- subsumes, 191 (Code 6.6.6)
- subsumes, 191 (Code 6.6.6)

subterm, 39 (Definition 2.2.5)  
 at address ( $\_@\_$ ), 97 (Definition 3.4.2)  
 subterm, 179 (Code 6.4.2)  
 subterm ordering ( $\triangleright$ ), 39 (Definition 2.2.5)  
 superterm, 39 (Definition 2.2.5)  
 suspensions, 123 (Comparison 4.2.16)  
 Swierstra, S. D., 10 (Contribution 1.1.1), 63,  
 91, 237  
 Sym, 159 (Code 6.2.1), 168 (Code 6.3.1)  
 Symbol, 168 (Code 6.3.1)  
 symbolic differentiation, 58 (Example 2.6.16)  
 symmetric, 32 (Definition 2.1.5)  
 syntactic extensions, 61  
 syntactic restrictions, 53 (Notation 2.6.2)  
 syntax, 13

## T

take from sequence ( $\backslash$ ), 37 (Notation 2.2.1)  
 $\tau_h$ , 51 (History 2.5.3)  
 Technische Universiteit Eindhoven, 4  
 term graph rewriting, 26, 96, 99 (Comparison 3.4.7)  
 term rewriting systems, 53 (Remark 2.6.3)  
 terminal node, 103 (History 3.4.12)  
 terminating, *see* SN  
 terms, 38  
 CRS, 52 (Definition 2.6.1)  
 CRSa, 142 (Definition 5.2.1)  
 CRSar, 145 (Definition 5.2.5)  
 $\lambda$ , 42 (Definition 2.3.2)  
 $\lambda_a$ , 101 (Definition 3.4.8)  
 $\lambda_X$ , 93 (Definition 3.3.18)  
 $\lambda_{\text{let}}$ , 110 (Comparison 3.5.7)  
 $\lambda_s$ , 91 (Definition 3.3.14)  
 $\lambda_\sigma$ , 93 (Definition 3.3.21)  
 $\lambda_\nu$  ( $\wedge\nu$ ), 86 (Definition 3.3.2)  
 $\lambda_x$ , 64 (Definition 3.1.2)  
 $\lambda_{xa}$ , 107 (Definition 3.5.1)  
 size, 38 (Notation 2.2.3)  
 $\text{\TeX}$ , 2  
 Text, 158 (Code 6.1.5), 168, 169 (Code 6.3.1),  
 175 (Code 6.4.1), 178 (Code 6.4.2)  
 TGR, *see* term graph rewriting  
 theory, 46 (History 2.4.2)  
 THESIS, 9  
 $\Theta$ , 44 (Notation 2.3.8)

Three Instruction Machine, 131 (Comparison 4.4.3)  
 thunks, 123 (Comparison 4.2.16)  
 TIM, *see* Three Instruction Machine  
 tm, 156 (Code 6.1.2)  
 Tofte, M., 20, 45, 239  
 Torkil Holms Fond, 4  
 total, 31 (Definition 2.1.3)  
 Toyama, Y., 112, 242  
 tr, 156 (Code 6.1.2)  
 tracing, 156 (Code 6.1.2)  
 Trans, 158 (Code 6.1.5)  
 trans, 158 (Code 6.1.5)  
 trans1, 158 (Code 6.1.5)  
 trans2, 158 (Code 6.1.5)  
 transformations, 116 (Principle 4.1.3), 203  
 (Code 6.9.8)  
 transformed, 128 (Comparison 4.3.5)  
 transition, 116  
 transition system, 116  
 transitive, 32 (Definition 2.1.5)  
 translation, 33 (Definition 2.1.10)  
 $\lambda\text{NF}/\lambda$ , 51 (Definition 2.5.4)  
 $\lambda_s/\lambda_x$ , 91 (Definition 3.3.15)  
 $\lambda_\nu/\lambda_x$ , 86 (Definition 3.3.5)  
 translation maps, 158 (Code 6.1.5)  
 trc, 156 (Code 6.1.2)  
 Treleaven, P. C., 22, 58, 96, 146, 234  
 triad machines, 132 (Comparison 4.4.3)  
 $\triangleright$ , *see* subterm ordering  
 TRS, 51, 53 (Remark 2.6.3), 54 (Example 2.6.7), 56 (Remark 2.6.11)  
 truncate, *see* take  
 Turing, A. M., 9, 20, 243  
 fixed point combinator ( $\Theta$ ), 44 (Notation 2.3.8)  
 Turner, D. A., 49 (History 2.4.10), 112, 142,  
 233, 243

## U

Ullman, J. D., 166, 233  
 UN, 35 (Definition 2.1.15)  
 xgc, 69 (Proposition 3.1.10)  
 unbrace, 171 (Code 6.3.2)  
 unification, 188 (Code 6.6.3)  
 unique normal forms, *see* UN



University of Oregon, 4  
 unravel ( $\Delta(-)$ ), 97 (Definition 3.4.1)  
 updating, 97 (Definition 3.4.2), 145 (Definition 5.2.6), 174  
 $\xrightarrow{v}$ , 86 (Definition 3.3.2)  
 URL, 18

## V

$\xrightarrow{v}$ , 48 (Definition 2.4.5)  
 Valuation, 175 (Code 6.4.1)  
 valuation, 55 (Definition 2.6.8)  
 value, 48 (Definition 2.4.5)  
 van Eekelen, M. C. D. J., 22, 27, 58, 96, 105  
   (History 3.4.13), 112, 131 (Comparison 4.4.3), 146, 234, 238, 240–242  
 van Oostrom, V., 4, 30, 51, 174, 181 (Code 6.4.4), 238, 240  
 van Raamsdonk, F., 4, 51, 174, 238, 240  
 Var, 159 (Code 6.2.2)  
 variable, 52 (Definition 2.6.1)  
   capture, 43, 66 (Remark 3.1.7)  
   clash, 43, 66 (Remark 3.1.7)  
 variable convention, 42 (Convention 2.3.4),  
   56, 123 (Remark 4.2.14)  
   for CRS, 56 (Convention 2.6.10)  
 Vars, 168 (Code 6.3.1)

## W

Wadler, P., 15, 110 (Comparison 3.5.7), 155,  
 233, 234, 237  
 Wadsworth, C. P., 11 (Contribution 1.1.1),  
 61, 146 (Comparison 5.2.8), 243  
 WCR, *see* LC  
 weak, 56 (Remark 2.6.11)  
 weak, 184 (Code 6.5.2)  
 weak CR, *see* LC  
 weak head normal form, *see* whnf  
 weak reduction, 47 (Definition 2.4.3)  
 weakly normalising, *see* WN  
 weakly orthogonal, 58  
 weakly regular, 58  
 weaklynonoverlapping, 186 (Code 6.6.2)  
 weaklyorthogonal, 191 (Code 6.6.4)  
 well-formed addresses, *see* sharing  
 wfa, 97 (Definition 3.4.2)

wfar, 145 (Definition 5.2.6)  
 whnf, 47 (Definition 2.4.3)  
 wind, 21  
 Wirth, N., 20, 243  
 WN, 35 (Definition 2.1.15), 185 (Code 6.5.3)  
 world wide web, 18  
 Wray, S. C., 131 (Comparison 4.4.3), 236

## X

$\xrightarrow{x}$ , 65 (Definition 3.1.4)  
 xdistribute, 195 (Code 6.7.3)  
 xeliminate, 196 (Code 6.7.4)  
 $\xrightarrow{xgc}$ , 65 (Definition 3.1.4)  
 ( $\xi$ ), 46 (Definition 2.4.1)  
 xintroduction, 193 (Code 6.7.2)  
 xtrans, 158 (Code 6.1.5)  
 Xy-pic, 2

## Y

Y, 44 (Notation 2.3.8)  
 Yoshida, N., 124 (Comparison 4.2.16), 243

## Z

( $\zeta$ ), 118