

University of London
Imperial College of Science, Technology and Medicine
Department of Computing

The Geometry of Implementation

(Applications of the Geometry of Interaction to language implementation)

Ian Craig Mackie

A thesis submitted for the degree of
Doctor of Philosophy of the University of London

28 September 1994

Abstract

This thesis aims to develop efficient implementation techniques for functional programming languages using novel technology arising from recent work on *Linear Logic*, the *Geometry of Interaction*, and *Optimal Reductions*.

The philosophy behind the implementations that we consider is that they should arise naturally from some underlying semantics of the language. We are then in a setting where correctness becomes a triviality, and semantic tools are at hand to assist directly with any optimisations, transformations, etc. that we may apply. In other words our interests are to implement the theory directly, rather than to find a theory that fits the practice; putting an engineering perspective on the theory.

We begin in Chapter 1 with general motivations for the use of the semantic tools that we will utilise, in particular the Geometry of Interaction. We go on in Chapter 2 to give basic definitions and background for the ideas used in this thesis.

In Chapter 3 we will look at translating the λ -calculus in to Linear Logic proof structures which will be the starting point of all our implementations. We then go onto *extracting* the result from these proof structures.

Chapter 4 takes a detailed look at the notions related to the Geometry of Interaction, namely Paths, Labels and Types.

Lafont's Interaction nets provide the first implementation of Linear Logic that we shall consider in Chapter 5. We give one very simple implementation of the λ -calculus as an interaction net, and work through the proofs of correctness using ideas from Girard's Geometry of Interaction. We address the problem of Lévy's theory of optimality too; however, we stress that this is *not* one of our goals. We are interested in efficient implementations of functional languages from a very practical point of view.

An alternative implementation is given in Chapter 6 which is based on data-flow. We show that we can set up a computational interpretation of the Geometry of Interaction that mimics cut-elimination in the λ -calculus. This chapter concludes by giving several very simple concrete implementations of this data flow on a standard

von-Neumann architecture. More specifically, we give a coding of the Geometry of Interaction in assembly language.

We conclude our ideas in Chapter 7 and suggest some further extensions to this work. In particular, we look at Hybrid systems, derived from combinations of the ideas presented in the two previous chapters.

Contribution

The work reported in this thesis is built upon a great many ideas arising from recent work in Linear Logic and Optimal Reductions, using Girard’s Geometry of Interaction — a dynamic semantics for Linear Logic.

The aim of this thesis is to clarify some of these mysterious ideas and try to use them in a practical way for the development of functional programming language implementations. The research reported here is very much incremental in nature, built upon the work of [GAL92], [DR93], [Lam90], [Gir89a], [AL93b], etc.; but examining the foundations of the subjects.

To make this a self contained thesis there is necessarily a considerable amount of background material that we need to cover. This is predominantly in Chapters 2 and 3, and the beginning of Chapters 4 and 5.

The main contributions of this thesis are outlined as follows:

- Chapter 3 gives codings of the λ -calculus and PCF in to Linear Logic proof structures develops a notation and systematically defines the standard translations.
- Chapter 4 presents the notion of a *path* in a Linear Logic proof structure. We extend the algebra of paths (called Λ^*) to include the constants required to code PCF, giving the algebra Λ_{pcf}^* . We will show some properties of this algebra and hint at some connections with Game semantics.
- Chapter 5 provides a new Interaction Net implementation of Linear Logic proof structures that turns out to be very simple; without a lot of the complication of extant Interaction Net implementations for Linear Logic. We extend these ideas for a pure typed functional programming language (PCF) and show how the ideas from the Geometry of Interaction can be used as a proof technique. This provides a sound basis for correctness.
- Chapter 6 shows a novel implementation technique for Girard’s Geometry of Interaction; the Geometry of Interaction Machine, in which we encode

PCF directly into assembly language and use results from Chapter 4 to give optimisations.

Throughout this work, we have tried to develop theory and practice hand-in-hand, producing prototype implementations of *all* the ideas presented in this thesis. The implementation is discussed in Appendix A.

Acknowledgement

My sincere thanks go to friends and colleagues of the theory and formal methods section at Imperial College who have all helped to provide a fruitful and enjoyable research atmosphere over the last few years. I am particularly grateful to Chris Hankin who has always shown a keen interest in my progress over the last seven years I have been at Imperial.

My work owes much to my supervisor Samson Abramsky. I have very much enjoyed working under his supervision; his knowledge and insights into the subject have always inspired me into wanting to know more.

It has been a great pleasure for me to share an office with Simon Gay for most of my PhD. Simon's understanding and explanations of so many aspects of the subject have been a great help to me. Additionally, he has always shown a keen interest in my work for which I am most grateful. Mark Dawson has always offered an ear to listen to my ideas and providing computing support. My work owes a great deal to the discussions I have had with Pasquale Malacaria and Radha Jagadeesan for which I am immensely grateful.

It is my great pleasure to thank Vincent Danos. I have learnt a great deal about the Geometry of Interaction from his writings and discussions.

Part of this work was done in Paris whilst visiting the LIX laboratory at École Polytechnique and the PARA group at INRIA Rocquencourt. My sincere thanks to Radhia Cousot and Jean-Jacques Lévy for making me so welcome and providing a productive environment — most of the details of Chapter 6 were developed during this visit. A very special thanks to Eric Goubault at École Normale Supérieure for making so many arrangements for this visit.

Over the last few years I have learnt a great deal from discussions with so many people who have shown interest in my work and provided me with motivation: T.K. Lakshman, Cosimo Laneve, Georges Gonthier and Laurent Regnier.

Roy Crole, Chris Hankin, Raja Nagarajan and Guy McCusker helped in the production of this thesis. Time has not allowed the inclusion of all their comments, but they have greatly improved the present document.

Many macros used in the $\text{\LaTeX}$ preparation of this thesis were developed by Paul Taylor. In particular, proof tree and diagram macros were used.

I am very happy to acknowledge the financial support from SERC in the form of a studentship, and the Esprit Basic Research Action project 6454 “CONFER” in the form of a bursary.

Finally, and most important, I thank my family for their support throughout which made this thesis possible.

This version of the thesis includes some minor changes recommended by my examiners Lincoln Wallen and Richard Bornat.

Contents

List of Figures	9
List of Tables	12
1 Introduction	15
1.1 The Geometry of Implementation	15
1.2 Linear Logic and the Curry-Howard Isomorphism	17
1.3 The Geometry of Interaction	20
1.4 Optimal Reduction	21
1.5 Thesis in outline	22
2 Background	25
2.1 Introduction	25
2.2 Lambda Calculi	27
2.3 Linear Logic	49
2.4 Linear Realisability Algebras	52
2.5 LRA's and Proof Nets	56
2.6 Discussion	57
3 λ-calculus in Proof Structures	59
3.1 Introduction	59
3.2 Translations	60
3.3 Properties of the translations	69
3.4 PCF in Linear Logic proof structures	73
3.5 Discussion	82

4	Paths, Labels and Types	85
4.1	Introduction	85
4.2	Proofs and Paths	87
4.3	Context Semantics	103
4.4	Other Notions of Paths	106
4.5	Dynamic Algebras	107
4.6	Relating Paths, Labels and Types	112
4.7	Discussion	115
5	A Lambda Evaluator Based on Interaction Nets	117
5.1	Introduction	117
5.2	Interaction Nets	119
5.3	Nets for the λ -calculus	132
5.4	Properties of reduction	139
5.5	Extensions	143
5.6	Analysis	153
5.7	Alternative reductions	159
5.8	Discussion	162
6	A Sequential Data Flow Machine	163
6.1	Introduction	163
6.2	Sequential Data Flow	165
6.3	The Geometry of Interaction Machine	167
6.4	Discussion of the implementation	181
6.5	Discussion	192
7	Conclusions and Further Work	195
7.1	General Conclusions	195
7.2	Interaction Nets	196
7.3	Geometry of Interaction Machine	198
7.4	Further work	198
7.5	Summary	201

<i>CONTENTS</i>	9
A GOI-Tools	203
A.1 Introduction	203
A.2 Interaction Nets	204
A.3 GOI Machine	204
A.4 Using the system	204
A.5 Availability	208
Bibliography	209

List of Figures

2.1	Reduction graph	31
2.2	Syntax: Linear $\lambda\sigma$ -calculus	46
2.3	Linear Logic	51
2.4	Syntax: Linear Realisability Algebras	52
2.5	Assignment of LRA's to Linear Logic proofs	55
5.1	Dynamics of Cut elimination for Linear Logic	126
5.2	Dynamics of Cut elimination for Linear Logic, continued	126
5.3	Dynamics of Cut elimination for Linear Logic, continued	127
5.4	Weakening rules for Linear Logic	127
5.5	Weakening rules for Linear Logic, continued	128
5.6	Interaction Net Agents for the λ -calculus.	133
5.7	Examples of graphs generated: I , K and S	136
5.8	Dynamics	138
5.9	Garbage Collection	139
5.10	PCF Dynamics	147
5.11	PCF Dynamics, continued	148
5.12	PCF Dynamics, continued	149
5.13	Extensions for arithmetic operations	151
5.14	List Interactions	152
5.15	A counter example for optimality	154
5.16	An example of how our algorithm gains	156
5.17	Commutative Conversion	161
6.1	Object code produced	177

6.2 Example trace of the Geometry of Interaction Machine 178

6.3 Example trace of the Geometry of Interaction Machine, continued 179

A.1 Syntax for GOI-Tools 205

List of Tables

3.1	$!D \multimap D$ translation	68
3.2	$!(D \multimap D)$ translation	68
3.3	$!D \multimap !D$ translation	68
3.4	$!A \multimap B$ translation	80
3.5	$!(A \multimap B)$ translation	80
5.1	Benchmark Results: Interaction nets	158
6.1	Benchmark Results: Basic Geometry of Interaction Machine	182

Chapter 1

Introduction

The aim of this thesis is to develop a basis for efficient, correct implementation of functional programming languages.

A suitable logic yields a framework for correctness; the associated semantics dictates dynamical aspects of the programming language and reduction techniques are crucial in addressing efficiency issues.

We make use of techniques arising from recent work on the following topics:

- Linear Logic, a logic developed by Girard [Gir87],
- The Geometry of Interaction, a semantics of Linear Logic, also developed by Girard [Gir89a],
- Optimal Reduction, a λ -calculus technique due to Lévy [Lev80] and Lamping's implementation [Lam90] of it.

Linear logic has precise accounting of resources, and is constructive. Hence it is suited for reasoning about correctness and efficiency. We believe that the Geometry of Interaction, because of its inherently dynamical nature, is the appropriate tool for semantics. We exploit the work of Gonthier [GAL92] as a guiding paradigm for insight into reduction strategies.

This research is an attempt to combine techniques from these rather disparate pieces of research to obtain methodologies that are immediately applicable to the implementation of functional languages.

1.1 The Geometry of Implementation

The philosophy behind the implementations that we consider is that they should arise naturally from some underlying semantics of the language. We are then in a

setting where correctness becomes a triviality, and semantic tools are at hand to assist directly with any optimisations, transformations, etc. that we may apply. In other words our interests are to implement the theory directly, rather than to find a theory that fits the practice; putting an engineering perspective on the theory.

Extant semantic paradigms for programming languages have indeed been considered for implementation before, for example the Categorical Abstract Machine [Hue90]. The semantic paradigm that we shall consider will be free from syntactical problems of calculi (for example substitution) while still capturing the operational behaviour, hence a genuine mathematical semantics of computation in which the dynamics is exposed. Such a notion has been proposed by Girard under the name of *Geometry of Interaction* in a series of papers [Gir89b, Gir89a, Gir88, Gir94].

Thus our aims in general terms are to generate a compiler technology for *functional programming languages* which does little more than implement directly the underlying semantics — the Geometry of Interaction. This thesis is about implementing this geometry; or, to put it another way, it presents *The Geometry of Implementation*. This methodology is somewhat different to that of Semantic Directed Compiler Generation as studied by, for example, Mosses [Mos79], which generates a compiler directly from some machine readable semantic description of a programming language. The resulting compilers produced in that work essentially compile all programs to a *sugared λ -calculus* which then has to be implemented by conventional means. Here we look at implementing the λ -calculus itself by using ideas from the underlying semantics.

Compiler technology for the core of a simple imperative language (for example a While language) is a solid, stable subject [ASU86]. This is by virtue of the language being not so far removed from any typical architecture. For declarative languages, however, it is much less stable, and new ideas and tools are being developed all the time. This is because the language is sufficiently removed from any architecture and compilation is generally described via some intermediate abstract machine. We aim to contribute to this thriving field by presenting some radical changes of perspective on what an implementation could be.

There are a number of established implementation techniques for functional programming languages. Broadly speaking these are based on graph rewriting [Pey87], stack manipulation, for example the SECD machine [Lan64], and buffered data-flow [FH88]. In this thesis we present two implementations of a simple functional programming language.

- First, building on Lafont’s work [Laf90], we present a very simple interaction net implementation which has solid foundation in the proof theory of Linear

Logic. This approach leads to a graph rewriting system which, although not optimal, captures sharing in the λ -calculus.

- Secondly, we present the *Geometry of Interaction Machine* which is a completely new implementation technique based on sequential data-flow, (without buffering), and has a solid foundation in the Geometry of Interaction interpretation of Linear Logic. This approach yields an implementation which has a very small and compact object code and run-time system.

1.2 Linear Logic and the Curry-Howard Isomorphism

Functional languages have their theoretical foundations in the λ -calculus; indeed it is the canonical form of such languages. The Curry-Howard isomorphism [How80] (see also [GLT89, Abr93]) establishes a tight relationship between intuitionistic propositional logic and the simply typed λ -calculus. The isomorphism can be explained by considering the term formation rules for the simply typed λ -calculus.

$$\frac{}{\Gamma, x : A \vdash x : A} \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x. t : A \rightarrow B} \quad \frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

which are called variable, abstraction and application respectively. If the terms are taken out of the above presentation we are left with the following, which is the natural deduction presentation of intuitionistic logic.

$$\frac{}{\Gamma, A \vdash A} \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \quad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

From this presentation one sees immediately that there is a correspondence between types of the λ -calculus and formulae of intuitionistic logic. Slightly more hidden is the fact that there is a correspondence between the terms of the calculus and the proofs of the logic. An analysis of the process of cut-elimination in intuitionistic logic and the normalisation process of the λ -calculus yields a further correspondence. Hence, taking programs to be terms from the λ -calculus, the full Curry-Howard isomorphism can be seen as:

$$\begin{array}{lll} \text{(functional) programs} & \sim & \text{proofs} \\ \text{types} & \sim & \text{formulae} \\ \text{normalisation} & \sim & \text{cut-elimination} \end{array}$$

and is known under a series of names including *formulae-as-types* and *proofs-as-programs*.

Linear Logic [Gir87] provides a refinement of intuitionistic logic in that the hidden *structural rules* of weakening and contraction are removed:

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B} \text{weakening} \qquad \frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B} \text{contraction}$$

The removal of these rules leads to a refined logic shown below, where $\multimap$ is the *linear* implication.

$$\frac{}{A \vdash A} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \qquad \frac{\Gamma \vdash A \multimap B \quad \Delta \vdash A}{\Gamma, \Delta \vdash B}$$

It is evident that this restricted logic makes clear the use of formulae in the proof; the slogan is “use all resources exactly once”. There is a Curry-Howard isomorphism too for this logic and a (linear) typed λ -calculus; λ -terms that use all their arguments exactly once.

Such a logical system is too weak to be of any practical use. To regain strength the weakening and contraction rules are re-introduced in a controlled way, not as (hidden) structural rules but as logical rules. This is achieved by introducing a modality ! (read “of course”) for formulae which indicates non-linear usage. Again there is a corresponding notion of a program for this logic. By introducing weakening and contraction as *first class* rules we arrive at a system where discarding and copying become explicit in programs.

Linear Logic has recently received great interest in many areas of computer science. It has provided new insights into many existing theories (for example optimal reduction [GAL92]), provided stepping stones for others (for example the recent solution for the full abstraction problem for PCF [AJM94, HO93]), and provided new theories and ways of reasoning in its own right, for example the Geometry of Interaction which is the subject of this thesis. It has also provided an approach for investigating issues in functional programming languages such as refined operational semantics, memory organisation and even the possibility of safe side-effects [Wad90].

This work is aimed at looking deeper into the notion of β -reduction in the λ -calculus by viewing it through Linear Logic. The λ -calculus is too abstract and its reduction steps too “big” to be of use directly as the basis of an implementation.

1. The λ -calculus does not provide any hint about the implementation of β -reduction. Indeed, the primary technique used to implement β -reduction,

namely explicit substitutions, already takes us out of the realm of the pure λ -calculus.

2. The λ -calculus does not give us information regarding argument use, in particular copying of arguments. Thus, one needs sophisticated techniques to extract this information, for example, abstract interpretation [AH87].

This motivates the use of proof structures of Linear Logic as a foundation for functional programming.

Linear Logic provides a decomposition of the functional arrow $A \rightarrow B$ into $!A \multimap B$:

- The type reflects the fact that functional application is a two phase process — a *linear* ($\multimap$) part that suggests a single usage of the argument, and a non-linear part reflected by the modality $!$ which allows multiple usage of the argument.
- This in turn induces a decomposition of the λ -calculus in which both the *abstraction* and *application* are decomposed into more primitive operations.
- Information is made explicit in the proof structure and in particular the progress of a substitution through a term appears as explicit steps in the cut-elimination process.
- Finally, as a consequence of the above, the linear types have more information about use/reuse than the intuitionistic types.

By looking deeper into reduction of Linear Logic proofs we can devise new reduction schemas in the λ -calculus. For example the notion of mixed call-by-value and call-by-name evaluation is quite natural, and the notion of *sharing* becomes explicit.

Hence our aim is to answer a question:

Can Linear Logic say anything about functional programming language implementation?

in the affirmative.

Functional programming languages based on Linear Logic have been proposed and investigated by a number of people, most notably Lafont [Laf87], Wakeling [Wak90], Holmström [Hol88] and also the present author. The only implemented system available is the Lilac system of the present author, reported in [Mac91, Mac94].

The basic idea proposed in that work was to program directly in proof terms for Intuitionistic Linear Logic (*the Linear Term Calculus*). This led to a fruitful paradigm in programming languages having mixed evaluation orders etc. However, the work has its limitations in that the “bookkeeping” information of the algorithm became larger than the algorithm itself, and the details of the intended computation were lost.

This work of this thesis will take a different perspective and code functional languages into Linear Logic directly. This frees the programmer from the extra burden of being aware that we are using Linear Logic proof structures as a basis of the implementation, but of course removes the insight that the programmer had about usage of parts of the program; these would now have to be performed automatically if we wished.

1.3 The Geometry of Interaction

Semantics has played a crucial rôle in the past by providing a tool for proving (static) properties of programs and implementations. A new paradigm in semantics developed by Jean-Yves Girard under the name *Geometry of Interaction* [Gir89a] can now allow the study of the dynamics of computation. The dynamics give a highly decomposed notion of a reduction step, which is free from syntactical problems such as substitution, and, as we shall demonstrate in this thesis, can be trivially implemented.

The philosophy behind this semantics is a notion of information flow through a program. The meaning of a program can be regarded as a set of information flows which are called *paths* in the program. As reduction proceeds one would expect that the information flows are invariant—no new information flows can be created, and no information flows can be lost. The meaning of a program is given by a set of paths, and computation is given by calculating the transitive closure of these paths. Girard has a useful analogy of thinking of these paths as electrical circuits—as reduction proceeds the electrical connectivity is preserved.

One of the main characteristics of the Geometry of Interaction program is that the semantics of a proof (program) is given as an operator on the Hilbert space ℓ^2 . In his first paper on the subject (*op. cit.*) Girard interpreted terms of system F [Gir71] (or the polymorphic λ -calculus [Rey74]) as operators in this space and showed that the derived operators are *nilpotent*; a notion of convergence. In his PhD thesis Vincent Danos proved that the property of nilpotency is strictly related to strong normalisation, in that the operator derived from a λ -term is a nilpotent operator *if and only if* the term is strongly normalising. So if nilpotency captures the meaning of terminating programs, what characterises pure functional programs, terminating

or not? In his second paper on Geometry of Interaction [Gir88] Girard proposed the notion of *weak nilpotency* (i.e. the weakest notion of convergence on ℓ^2) and showed that fixed point operators à la ML satisfy this property. The result of weak nilpotency for the pure λ -calculus was proved by Pasquale Malacaria and Laurent Regnier in [MR91]. The proof is a corollary of the main result of that paper namely the “acyclicity theorem” which states that every cyclic path in a pure net cannot be repeated consecutively. This result of a more syntactic flavour has been later applied by Danos and Regnier in order to show the confluence of the virtual reduction.

What is much more significant is that these paths can be represented by a composition of elements of an algebra (Λ^*). An equational theory on this algebra gives a notion of *normal form* for paths. Therefore each thread of computation can be reduced to normal form independently of the rest of the paths; an *asynchronous* reduction. Each of the rewrite rules of the equational theory are *local* operations in that they only affect the part of the path that is being reduced. The semantic paradigm therefore leads to a local and asynchronous device that can be implemented.

For deterministic computation of function programs at ground type there is only one (unique) thread of control, hence only one path to compute—the execution path. It is this class of languages that we are interested in implementing in this thesis.

Related work has been done by Vincent Danos and Laurent Regnier [DR93] who have developed the theoretical foundations of reduction in the λ -calculus based directly on ideas from Girard’s Geometry of Interaction. Their notion of *virtual reduction* is based on computing the set of paths of a λ -term, and yields a local and asynchronous implementation of β -reduction. Although not mechanised, their implementation is based on a graph representation of the set of paths. Reduction proceeds by rewriting the graph in a way dictated by the Geometry of Interaction. Our work will differ from this in that we will *not* have a graph representation of the set of paths, but actually compile the graph and compute the execution path in a more direct way.

1.4 Optimal Reduction

In 1980 Jean-Jacques Lévy [Lev80] introduced a notion of optimal reduction for the λ -calculus based on the idea that a reduction of a λ -term to normal form should do the *least number of β -reductions*; a notion that we will review in the next chapter. Hence there is a slogan: do as few β -reduction steps as possible, which is the notion, theoretically, of *work* to be done by an implementation.

There was then a “big gap” in the history of optimal reductions until, in 1990, John Lamping [Lam90] and Vinod Kathail [Kat90] introduced an algorithm to implement the theory. This gap perhaps indicates just how complex this problem is. To

get more sharing one needs to develop implementations which essentially “switch” between different orders of evaluation. The theory of optimality has been successfully implemented by [Kat90, Lam90]. Both these algorithms make use of partial sharing of redexes.

The work of Gonthier, Abadi and Lévy [GAL92] made a beautiful connection between optimal reduction and the Geometry of Interaction by observing that the sharing combinators used by Lamping were in fact strongly related to the notion of path computation in the Geometry of Interaction. This insight lead to a significant clean-up of the presentation of Lamping’s algorithm.

The connection between Lamping’s algorithm and virtual reduction was exposed in [ADLR94], where many notions of paths are shown to be equivalent, and in particular, the notion of path computation and Lamping’s graphs are shown to be equivalent.

In a more practical setting the notion of work being a β -reduction step seems less clear to be the right notion. In particular the implementation of such a reduction step is certainly not a constant time operation. We feel therefore that all these ideas warrant study in a practical setting where a notion of *work* should be based on a the number of computation steps.

1.5 Thesis in outline

We propose to develop in this thesis some ideas related to the above with an *engineering* perspective. We try to extend some of the ideas to cover a more practical programming language, and aim to keep efficiency and pragmatics in the foreground. We propose to add to this wealth of research some new ways of implementing functional programs. In particular, our main aims in global terms are:

1. To understand more clearly the notion of a β -reduction in the λ -calculus by distilling the essence of the work outlined above.
2. To develop an implementation of functional programming language concepts in the context of Linear Logic, the Geometry of Interaction, and sharing graphs for the λ -calculus. In particular, we will examine some of the problems with Lamping’s algorithm and try to develop something new which will overcome these deficiencies. We stress that optimal reduction will *not* be an overriding theme of this thesis, but we will appeal to the theory for new techniques.
3. To develop a theory and an implementation arising directly from the Geometry of Interaction. It is this work that we see as the main thrust of our research,

providing very novel ideas in the world of compiler technology for functional programming languages.

We will concentrate on *sequential* implementation techniques, however, although not stated explicitly throughout the thesis, we will keep at the back of our minds the ideas of concurrent techniques. There is also another overriding constraint in that we demand that an implementation consists of *all* the machinery required to execute a program; so it is not assumed that there exists any external *discarding* (garbage collection) or *copying* mechanisms. Hence everything is built in and local to our implementations and the cost of computation can be calculated in a uniform way.

The thesis will begin by focussing on the untyped world for as long as possible (i.e. as long as the results hold!). We will then switch to a typed setting when we start to look at PCF; for which all the previous results will still hold.

The chapter structure of the rest of this thesis is as follows:

Chapter 2: Background

In this chapter we give basic definitions for the ideas used in this thesis. There is very little original work here; it is included to make the thesis reasonably self contained. We will review notations which we will use for the λ -calculus, sharing and Linear Logic.

Chapter 3: The λ -calculus in Linear Logic proof structures

Translations of the λ -calculus in to Linear Logic proof structures is taken for granted in many pieces of work on the λ -calculus and Linear Logic. This chapter will thoroughly describe how to translate the λ -calculus in to Linear Logic proof structures, and we will show some elementary properties of the translations.

Chapters 5 and 6 are the main thrust of this thesis where we show how to *extract* the “result” from these proof structures; first by performing a notion of cut-elimination, then a notion of extracting the result *without* rewriting the proof structure using ideas from the Geometry of Interaction and data-flow.

Chapter 4: Paths, Labels and Types

The foundation for all our work is the Geometry of Interaction. In this chapter we will review the notions that we will use, and extend the theory to cover the programming languages that we will study. We show properties of this interpretation which will lead to optimisations of our data-flow implementations in Chapter 6.

Chapter 5: A lambda evaluator based on Interaction nets

Lafont's Interaction nets provide the first implementation of Linear Logic that we shall consider. We give one very simple implementation of the λ -calculus as an interaction net, and work through the proofs of correctness using ideas from Girard's Geometry of Interaction. We address the problem of Lévy's theory of optimality too; however, we stress that this is *not* one of our goals. We are interested in efficient implementations of functional languages from a very practical point of view.

Chapter 6: A sequential data flow machine

Data flow mimics cut-elimination. To make this statement more precise we appeal to the proof technique used in Chapter 5 to give an implementation directly. Basically we have a *fixed* network and a *single* token that traverses this structure. The token travels a path that is *invariant* to the cuts in the proof.

This chapter concludes by giving several very simple concrete implementations of this data flow on a standard von-Neumann architecture. More specifically, we give a coding of the Geometry of Interaction in assembly language.

Chapter 7: Conclusions and further work

Finally, we conclude and suggest some further directions for this research programme. We hypothesise that these ideas can be extended and used in real implementations.

Appendix A: GOI-Tools

Throughout this work, we have tried to develop theory and practice hand-in-hand, producing prototype implementations of *all* the ideas presented in this thesis. The implementation is discussed in this appendix.

Chapter 2

Background

This chapter presents the calculi used and sets up the notation for the rest of the thesis.

2.1 Introduction

It is widely acknowledged that the λ -calculus is the theoretical foundation of functional programming languages. It is less clear, however, that the λ -calculus provides the right setting for implementation techniques for such languages. A wide variety of techniques have been used in practical implementations, including combinator reduction [Pey87], the categorical abstract machine [Hue90], and data flow [FII88]; but they are all based on particular models rather than on the proof theory of the λ -calculus itself. In contrast, our thesis is that the λ -calculus can be used much more directly as the basis for functional language implementations. Our approach thus has more in common with Wadsworth's graph reduction algorithm [Wad71, Chapter 4] than with the techniques mentioned above. Its main novelty is a foundation in the theory of linear logic [Gir87]. We will present implementations of the λ -calculus based first on Lafont's interaction nets [Laf90], which generalise the proof nets of linear logic, and then on Girard's Geometry of Interaction, particularly the recent notion of path [ADLR94]. Because the resulting implementations have a firm foundation in the proof theory of linear logic, correctness proofs are greatly simplified; furthermore, program transformations and compiler optimisations can be treated in the same framework.

With this clear view in mind we now embark on our programme of implementing the λ -calculus. In this chapter we begin by presenting the λ -calculus which we take as standard. We also review the different notions of evaluation order that one can impose upon the syntax. We will then review issues regarding the implementation

via *graph reduction* where *sharing* will be our focal point. Lévy’s theory of optimal reduction will be presented as a general theory of maximal sharing.

The most significant disparity between the fundamental formulation of the λ -calculus and the key features of its implementation arises from the operation of substitution. Substitution is usually defined formally through a meta-theoretic operation on the terms of the λ -calculus and β -reduction is defined using substitution. From the point of view of implementations, this results in a grave over-simplification of the concept of β -reduction, and does not allow for reasoning about the low level details of such computations: one such low level detail is that of *sharing*, the subject of this thesis.

To try to dig deeper into the problems of reduction we consider a calculus where we make the operation of substitution explicit. We remark at this point that this is *not* a version of the $\lambda\sigma$ -calculus considered by, for example, [ACCL91] which is founded on the notion of de-Bruijn indices. We avoid consideration of such a calculus which seems, thus far, to be a very complicated piece of machinery—there is no proof of Strong Normalisation for example. The calculus that we present is simply the λ -calculus with the *standard* notion of substitution made part of the syntax: in particular we keep variable names. Making this operation explicit in the calculus allows us to keep track of the progress of a substitution through a term during reduction, and provides a unified language where we can talk about the full details of a reduction.

Going with the view that “the more explicit the better”, the calculus we present will also contain syntactic constructs representing *copying* and *discarding* of arguments. We remark that there is nothing deep in this calculus; there are trivial translations for the λ -calculus to this calculus which are based on simple notions of variable counting. The reason for its introduction is that we can then give a more refined notion of pushing a substitution into a term—the notion of copying and discarding a substitution will become explicit during the reduction.

Finally, we give our presentation of Linear Logic and a proof calculus which which we are going to consider for implementation. We see this as a calculus which exposes the true structure underlying the λ -calculus.

The sequence of calculi that we will present provides a natural sequence of decompositions of the λ -calculus. Each calculus that we present subsumes the λ -calculus, and exposes additional structure. It is the final calculus (Linear Logic proof structures) that we propose to use for our implementations of the λ -calculus; these are reported in Chapters 5 and 6.

2.2 Lambda Calculi

2.2.1 The Classical λ -calculus

The starting point to all our work is the pure λ -calculus. We assume familiarity with this calculus, and here just fix the notation. We refer the reader to [Bar84] for the notation and [Han94] for a more gentle introduction.

Definition 2.2.1 The λ -calculus is built up of terms from the following syntax.

$$M ::= x \mid (\lambda x.M) \mid (MN)$$

which are respectively named *variables*, *abstractions* and *applications*. There are generally accepted conventions that application associates to the *left* and abstractions to the *right*. Abstractions of several variables can also be abbreviated to $\lambda \vec{x}.M$. We will follow all these conventions without further mention. $\diamond$

We refer the reader to the literature for notions of contexts ($C[\]$), Free Variables (fv), bound variables, the set of sub-terms of a term M ($sub(M)$) and substitution which we shall write as $M[N/x]$. We use the notations $M \equiv N$ and $M \equiv_\alpha N$ if M and N are syntactically identical and syntactically identical up to change of bound variables respectively. We will also adopt the *Variable Convention* of [Bar84] — all bound variables are chosen to be different from free variables.

Reduction of terms is defined by the β -rule:

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

which can take place in *any* context, i.e.

$$\frac{M \rightarrow N}{C[M] \rightarrow C[N]} \text{ context rule}$$

There are several different ways of reducing a term which can be obtained by restricting the context rule. For our work we need the following:

- A reduction is called *leftmost* if the redex being contracted is textually to the left of all other redexes.
- A reduction is called *outermost* if the redex being contracted is not contained in any other redex.
- A reduction is called *innermost* if the redex being contracted contains no other redex.

Recall that the *normal forms* of the theory are exactly the terms containing no sub-term of the form $(\lambda x.M)N$. However, practical implementations of functional programming languages stop some way short of this. Two ubiquitous notions of an *answer* are Head Normal Form and Weak Head Normal Form.

Definition 2.2.2 A term M is in *head normal form* if it is of the form:

$$\lambda x_1 \dots x_n. x M_1 \dots M_k \quad n, k \geq 0$$

where we call the variable x the *head variable*. The terms $M_1 \dots M_k$ may be any terms, and in particular need not be in any normal form. $\diamond$

Definition 2.2.3 A term is in *weak head normal form* if it is one of the following forms:

- $\lambda x. M'$
- $x M_1 \dots M_n \quad n \geq 0$

where $M_1, \dots, M_n$ are terms in weak head normal form, and M' is any term. $\diamond$

Hence weak head normal forms arise just when we *do not* evaluate under a λ , and head normal forms arise by experimenting *just inside* a λ . Each notion of reduction has its own distinctive character in the theory: head normal forms play a crucial rôle in *computability theory* associated with the λ -calculus, and weak head normal forms have provided a canonical notion of an *answer* in almost all implementations of functional programming languages. The former notion of answer is generally called the *classical* theory and studied at depth in [Bar84], and the latter has been studied in [Abr90] under the name *the lazy λ -calculus*.

Having defined various notions of *answers* in the theory, we now show that there are many different ways of getting there.

Call-by-name

The first reduction scheme that we consider is the simplest to explain. We simply allow the β rule to be applied to arbitrary sub-terms:

$$(\lambda x.M)N \rightarrow M[N/x]$$

following the *leftmost-outermost* reduction strategy until we reach a weak head normal form.

We make the following observations.

- Redexes occurring inside N may get copied by the action of substitution if x occurs more than once in the term M .
- If x does not occur inside M , then we have not wasted computation time on evaluating N before making the substitution.

Call-by-value

The second notion of reduction is slightly restricted by the fact that the β rule is only defined when the argument has been evaluated to some extent:

$$(\lambda x.M)V \rightarrow M[V/x]$$

and following the *leftmost innermost* reduction strategy until we reach a weak head normal form.

Hence V , the set of values, are weak head normal forms.

$$\begin{array}{lcl} V & ::= & \lambda x.M \\ & | & xV_1 \dots V_n, n \geq 0 \end{array}$$

We make the following observations.

- Since this notion of reduction forces *some* reduction of the argument before it is substituted, this work will not be duplicated later. However, there may still be redexes within the argument that get copied since we only evaluate arguments to weak head normal form.
- The argument may be discarded by substitution, in which case any computation on the argument was wasted.

During reduction in the λ -calculus terms can be copied and discarded. The consequence of this is that neither of the above reduction schemes seems reasonable on its own.

- A call-by-value reduction order should be imposed so that we avoid excess copying of redexes.
- A call-by-name reduction order should be imposed so that we do not waste computation in the evaluation of unused arguments.

The reduction strategies considered thus far are known as *copy based* in that terms are copied textually during reduction. A consequence of this is that redexes may be

copied during reduction since there is no means of *sharing* any computation. An alternative approach, known as *environment based*, is where we store the results of computations so that they are accessible to the remaining computation. One way of implementing this notion is *graph reduction* which will become a central theme of this thesis. For the moment we will stick with the current notation and present a reduction strategy which can be seen as an attempt to rectify the conflict between the two evaluation orders that we have seen so far.

Call-by-need

The final notion of reduction that we consider is a refined version of call-by-name which incorporates *sharing*. For this we introduce an environment env which the reduction is performed relative to. The environment here is simply a structure containing assignments of values to names. β -reduction is now defined as:

$$(\lambda x.M)N \text{ env} \rightarrow M \text{ env}[x = N]$$

where $\text{env}[x = N]$ is our notation for updating the environment.

Reduction continues by evaluating M in the new environment. When the value of x is *needed* in the computation we first evaluate N to weak head normal form and update the environment before copying the value associated to x to the term. Hence if x is accessed a second time it will benefit from using the *already evaluated* value of x .

Again, we make some elementary observations.

- If N is not used in the computation then there is no waste in computation since evaluation only begins when a value is needed.
- As with call-by-value, this strategy is not totally free from duplication of redexes since we only evaluate to weak head normal form—redexes within the term will still be copied.

Remark 2.2.4 The observations made above on the various evaluation orders *all* deal with the notion of substitution. We remind the reader of our desire for a calculus where the notion of substitution is made explicit so that there are finer tools available with which we can talk about this issue. $\diamond$

The above discussion is intended to provide the definitions and intuitions of notions for reduction in the λ -calculus. It is not meant to be encyclopedic, and we refer the reader to the literature for a more complete treatment of these delicate notions.

When reducing a λ -term to normal form, one would like to share as much work as possible. Lévy showed that order of evaluation is not good enough and no matter which order of evaluation is used there are always terms which could be reduced with less work using a different order of reduction. Indeed for any given term there is no simple order of evaluation that will suffice. In a sense, this is saying that sharing is an *orthogonal* issue to evaluation order; this certainly is not the case with, for example, (fully) lazy implementations where the aim is to get as much sharing as possible using a leftmost strategy. To get more sharing, one needs to develop implementations which can switch between orders of evaluation.

Example 2.2.5

The *reduction graph* shown in Figure 2.1 makes these comments a little clearer. We consider the term $(\lambda x.xI)(\lambda y.(\lambda z.zz)(yt))$, and write L for a *leftmost* reduction and R for a *rightmost* reduction. The unnamed reductions are when no choice can be made, i.e. there is only one redex. We begin in the top left-hand corner of the diagram, and the normal form in the the bottom right.

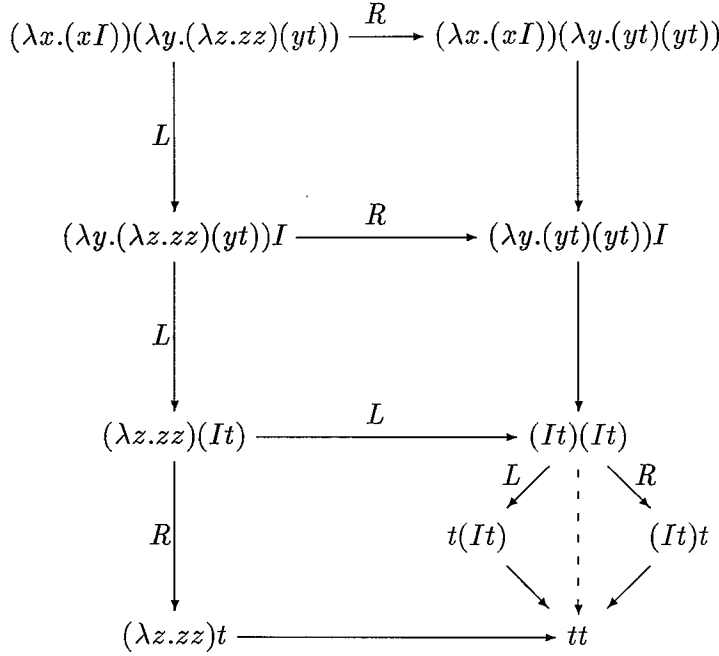


Figure 2.1: Reduction graph

We see from this example that both leftmost and rightmost strategies will duplicate the It redex. However, if we trace this redex back in the graph we see that both occurrences originate from a common source, (the yt potential redex), and could have been shared by permuting the reduction order. This is the fundamental observation by Lévy that led to his theory of optimal reduction; capturing residuals modulo

permutation of reduction. The dashed line indicates the redex that should be shared to give us the least number of β -reductions. This example indicates that we need to share not only *sub-expressions* but also *shared contexts* or *closures*. $\diamond$

2.2.2 Graph reduction

The graph representation of the λ -calculus is an ideal way of indicating sharing. We will recall the salient features of Wadsworth's algorithm [Wad71, Chapter 4].

First some notation: A graph is built up from a set of nodes including $@$ and λ which are the representations for application and abstraction respectively. Nodes are connected by vertices of the graph, and we will call the point of connection of a vertex to a node the *port* of the node.

The following diagrams show the application MN and the abstraction $\lambda x.M$.



For the graph rewriting rules we will follow the presentation given in [Pey87], but with the additional feature that sharing is represented explicitly with *fan-in* nodes.

A fan-in node is a representation of sharing in a graph. We will draw these as follows:

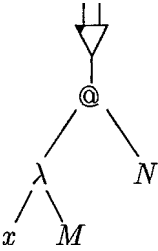


If there are two pointers to a node in a graph then we will use the above notation to bring the two pointers together so that each node in the graph has at most one vertex at the same port. Note that one side of this node is marked and the other left blank. This notation serves as a method of distinguishing each copy, and will play a significant rôle shortly.

The significant features of graph reduction are outlined below. We will assume some familiarity with graph reduction and just present an overview.

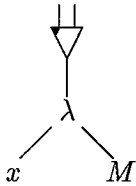
- A redex $(\lambda x.M)N$ may be shared, i.e. two pointers are directed at the root of

this redex:



so we *should* physically overwrite the root of the graph with the result $M[N/x]$ so that the computation is not duplicated by each copy; thus sharing the computation. We emphasise that we *should* do this for efficient implementation, but it is not necessary for the correctness.

- An abstraction $\lambda x.M$ may be shared:

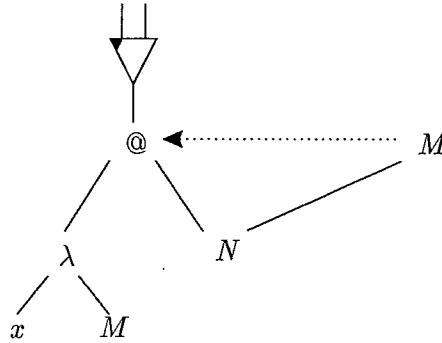


so we *must* construct a new instance of the whole body of the abstraction when we reduce the graph.

We emphasise that we *must* do this for the correctness of the implementation so as to avoid substituting the binding variable x which may be substituted by a different term in the other copy. Note in particular that all redexes in this term will be copied—we thus need to traverse the entire structure of the term and its substitutions and copy them. We remark also that it may be the case that this abstraction is used only once and the copying was needless—the fact that there are two pointers to the term does not imply that both copies are actually needed in the computation. However we could identify this situation by using static analyses (abstract interpretations [AH87]).

- To avoid excess copying, we *should* substitute *pointers* to the argument for the formal parameters rather than copying them. The following diagram shows that we copy the term M , substitute pointers for the variable x to point to N

and finally overwrite the root of the redex with the result, as indicated by the dashed line.



From the above point we are forced to copy M , the body of the abstraction, before performing the substitution. We must now traverse the structure of M and substitute pointers to the argument N for each free occurrence of the variable x . It should be noted that this is not a trivial operation and requires a substantial amount of computation.

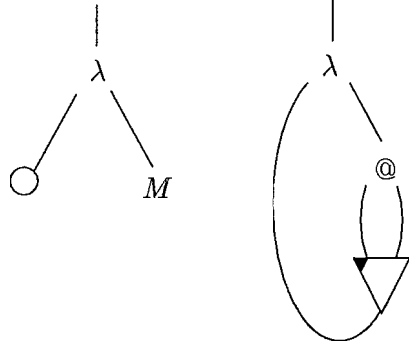
It is this final point that generates sharing (fan-in nodes) which arises when there is more than one occurrence of the variable x in M and requires that we have a *graph* data structure rather than a tree.

The second point above clearly indicates a weakness in capturing sharing during graph reduction, and is the reason that we do *not* get *optimal reduction* (to be defined shortly). We remark also that there is a weakness with respect to additional problems such as indirection nodes and locating the next redex during graph reduction. We refer the reader to the literature (see for example [Pey87]) for more details on these issues.

Having identified the problem we are in a fortunate position, courtesy of Lamping and Kathail, to give the solution. What is required is a mechanism for contracting all redexes in a term when a copy is made, postponing copying potential redexes, and only copying when needed. Lamping in [Lam90] identified the solution to this problem by introducing the dual of the fan-in node: the *fan-out* node. (These are just fan-in nodes drawn upside-down.) By using fan-in and fan-out nodes we can represent *partial applications* (or *closures* or *sub-expressions and their substitutions*).

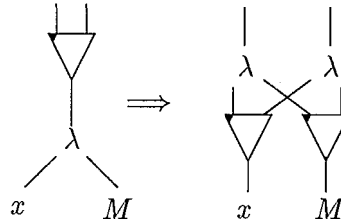
First there is a slight modification in the graph representation. We will draw abstractions $\lambda x.M$ as a graph as shown previously but connecting the occurrences of the variable x in M to the binding λ . If x occurs more than once in M , then we make this explicit by using fan-in nodes; if x does not occur in M then we make

this explicit by placing a *plug* at the end of the binding variable. The following diagrams show the representation of $\lambda x.M$ where x does not occur in M and the λ -term $\lambda x.xx$ where a fan-in node is used.



Fan nodes now become part of the rewriting system and they can be seen as building into the rewriting system a previously extraneous piece of *copying* machinery. The intuition is that the fan nodes perform local copying and traverse the structure in such a way as to never copy a redex. Hence sharing nodes now become, (rather schizophrenically), copying nodes.

Fan-out nodes are *created* by the following variant of the second rule of Wadsworth's algorithm, where a fan-in node is pushed into the body of the abstraction, and a fan-out node is created to connect two different bindings to the lambda:

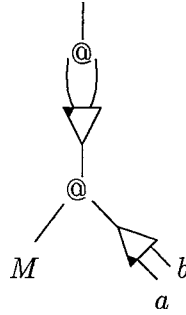


Here we see the term M , the body of the lambda, shared with two possible bindings for x . The different bindings are distinguished by the markings that we put on the fan nodes. We remark that each lambda node connects consistently to a marked or non-marked port of the fan nodes. For example the left-hand λ connects to the marked ports of each of the two fan nodes.

The process of a fan node moving through the graph copying only at a local level is crucial to the idea of partial sharing. In particular, they *propagate* through a term in such a way that redexes are not copied.

To give an example of how partial sharing works we consider the expression $(\lambda x.(xa)(xb))(\lambda x.Mx)$ which will reduce to $(Ma)(Mb)$ after a number of reductions.

We can represent this reduced term by the following graph:



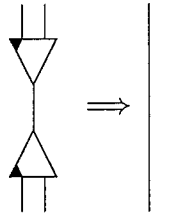
which shows the partial sharing of the “ M –” redex: Ma and Mb are represented as a *single* redex. Hence graph reduction will share all redexes in M .

The use of the marks on the fan nodes now become apparent if we try to read a term out of the above graph. The general idea is that when we traverse the structure of the graph to try to flatten it into a term we will enter and leave *shared* parts of the graph. The purpose of the marking is to indicate which copy of the shared part we are currently reading out.

With respect to the diagram above, if we enter the marked edge of the fan-in node we can flatten the graph to a term M – and then we must flatten the argument—either a or b . The fan marks state exactly which one of these is the right one for this copy—we select a since this is connected to the marked port of the fan-out node; corresponding to the marked port of the fan-in node.

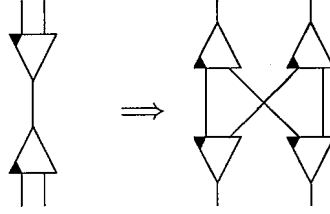
Thus far the concept has been quite simple since we introduced the fan-in node and fan-out node in pairs—there is a balancing notion of these nodes so that we know how to read out a term from the graph.

Fan nodes should be regarded as copying a λ -body for which there is a notion of *depth* (cf. de Bruijn indices [deB72]) or level of environment that these nodes are operating at. This begs the question as to what should happen when two fan nodes meet head on. There are exactly two possible cases: either the fan nodes are operating at the same level of the environment, in which case their job in copying is complete and they should cancel each other out:



or they are operating at different levels of the environment, in which case they

should copy each other:



The second case arises because we allow fan nodes to enter an environment not only from the λ but also from the free variables of a term. This is the problem associated with explicitly connecting the binding variable to the occurrences in the term.

Lamping introduced additional nodes to mark the free variables of the environment, and indices on nodes so that it is possible to distinguish fan nodes operating at different levels of the environment. We refer the reader to [Lam90] and [GAL92] for a presentation of these operators and more details of how they work.

The brief description of Lamping's algorithm given above is meant to be an overview, and hopefully the reader will gain further insight into these intuitions by the end of this thesis. We will also make clearer the notion of *levels of environment*.

2.2.3 The Theory of Optimality

Optimal reduction is a characterisation of which redexes should be shared. We begin with recalling some basic notions about reductions. The following sequence of definitions is essentially taken from Lévy's thesis [Lev78].

Residuals

Definition 2.2.6 Let $M \xrightarrow{R} N$ be a one step reduction, where R is the contracted redex in M . Let S be another redex in M . The *residuals* of S with respect to this reduction, written as S/R , is a multiset of redexes in N defined as follows by analysing the relative position of S with respect to R :

1. If R coincides with S then $S/R = \emptyset$
2. If R and S are disjoint terms, then $S/R = \{S\}$
3. If S is strictly in R , (i.e. $S \in \text{sub}(R)$ and $S \not\equiv R$), then there are two sub-cases. Let $R = (\lambda x.P)Q$, then
 - (a) $S \in \text{sub}(P)$, then $S/R = \{S[Q/x]\}$
 - (b) $S \in \text{sub}(Q)$, then $S/R = \underbrace{\{S, \dots, S\}}_n$ where n is the number of occurrences of the variable x in P .

◇

Hence residuals are a notion of tracing redexes during a reduction of a term.

Labelled λ -terms

Keeping track of residuals during a reduction becomes tedious using the above definition. We now show a notion of *labelled* λ -terms where the labels will serve as a tracing mechanism. Basically labels are a technical tool ideal for the study of residuals.

We present Lévy's original labelled λ -calculus, and refer the reader to [Klo80] for a detailed discussion on the various alternative notions of labels, and proofs of equivalence.

Definition 2.2.7 (Labels)

Let $\mathcal{L}_0$ be an infinite set of letters (our atomic labels). We define a set $\mathcal{L}$ of strings formed on $\mathcal{L}_0$ as follows:

$$\begin{aligned} a \in \mathcal{L}_0 &\implies a \in \mathcal{L} \\ \alpha, \beta \in \mathcal{L} &\implies \alpha\beta \in \mathcal{L} \\ \alpha \in \mathcal{L} &\implies \bar{\alpha} \text{ and } \underline{\alpha} \in \mathcal{L} \end{aligned}$$

◇

We define a set of *labelled* λ -terms on $\mathcal{L}$ as follows:

Definition 2.2.8 Let α be a label in $\mathcal{L}$, then we have the following labelled terms Λ' :

$$\begin{aligned} x \in V &\implies x^\alpha \in \Lambda' \\ M \in \Lambda' &\implies (\lambda x.M)^\alpha \in \Lambda' \\ M, N \in \Lambda' &\implies (MN)^\alpha \in \Lambda' \end{aligned}$$

◇

Before we define labelled β -reduction, we define the notions of label substitution and term substitution.

Definition 2.2.9 (Label substitution)

$$\begin{aligned} \alpha \cdot x^\beta &= x^{\alpha\beta} \\ \alpha \cdot (\lambda x.M)^\beta &= (\lambda x.M)^{\alpha\beta} \\ \alpha \cdot (MN)^\beta &= (MN)^{\alpha\beta} \end{aligned}$$

◇

Definition 2.2.10 (Term substitution)

$$\begin{aligned}
x^\alpha[N/x] &= \alpha \cdot N \\
y^\alpha[N/x] &= y^\alpha \\
(\lambda y.M)^\alpha[N/x] &= (\lambda y.M[N/x])^\alpha \\
(MP)^\alpha[N/x] &= (M[N/x]P[N/x])^\alpha
\end{aligned}$$

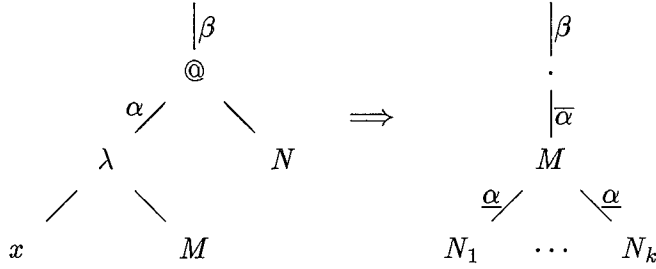
◇

Definition 2.2.11 We can now define the labelled β -reduction as follows:

$$((\lambda x.M)^\alpha N)^\beta = \beta \bar{\alpha} \cdot M[\underline{\alpha} \cdot N/x]$$

◇

This can be simply visualised by considering graphs of a term, and understanding the labels as labelling the *edges* of the graph.



The underline and overline structure of the label indicate a *direction* for the cut edge. The cut edge, which was a *shared* edge in the original graph has now been split in the resultant graph and it is the labelling that indicates the direction. Note that each copy of N gets the same *outer* label.

Example 2.2.12 We include two examples to give the reader a flavour of labelled reduction, and the kind of labels generated.

1. The first example is very simple: $(\lambda x.x)(\lambda x.x)$. The following shows the original labelled term and the normal form after one labelled β -reduction.

- $((\lambda x.x^a)^b(\lambda x.x^c)^d)^e$
- $(\lambda x.x^c)^{\bar{e}\bar{a}b\bar{d}}$

2. The reader will enjoy checking the second example which is included to show the kind of labels that can be generated during reduction. The term that we

consider is *SKII*. The following shows several “snap-shots” of the reduction.

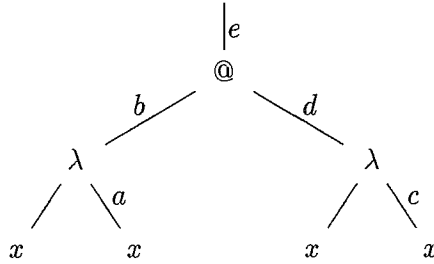
- $(((((\lambda x.(\lambda y.(\lambda z.((x^a z^b)^c(y^d z^e)^f)^g)^h)^i)^j)(\lambda x.(\lambda y.x^k)^l)^m)^n(\lambda z.z^o)^p)^q(\lambda z.z^r)^s)^t$
- $(((((\lambda x.(\lambda y.x^k)^l)^{a\bar{j}m}(\lambda z.z^r)^{b\bar{q}n\bar{j}ihs})^c((\lambda z.(\lambda z.z^r)^{o\bar{q}n\bar{j}ihs})^{d\bar{n}\bar{j}ip}(\lambda z.z^r)^{e\bar{q}n\bar{j}ihs})^f)^{t\bar{q}n\bar{j}ihg})$
- $(\lambda z.z^r)^{t\bar{q}n\bar{j}ihgca\bar{j}m\bar{l}ka\bar{j}mb\bar{q}n\bar{j}ihs}$

◇

Definition 2.2.13 The *degree* of a redex is the label of its abstraction part $degree((\lambda x.M)^\alpha N) = \alpha$. ◇

The Church-Rosser theorem for the labelled λ -calculus [Lev78] states that we capture residuals modulo permutation of reductions. In other words the labelling provides a methodology orthogonal to the reduction order to identify which redexes should be shared.

In Chapter 4 we will look more deeply into the structure of labels. Specifically we will recall a result from [AL93b] that says that the label generated in a term after reduction gives a notion of a *path* in the original term. We can show this quite easily using the first example in 2.2.12 above; using the notion of labelling a graph with the labels of the calculus. The graph representation of this example is given by:



The label generated in the normal form for this term is given by $e\bar{b}a\bar{b}d$. We should read this from left-to-right which gives a navigation around the above graph. Hence, we start at the root edge e , travel along the cut edge b in the positive direction (indicated by the overline structure of the label), then into the body of the λ along edge a to the variable x . At this point, recalling that the variable x is connected to the binding lambda, we return along the cut edge b in the negative direction (indicated by the underline structure of the label). Finally, we enter along d which is the argument of the application. Hence the outer label of the reduced term indicates a path in the original term to result of the computation.

The notion of a path in a term is a central notion in this thesis and we will see it arising throughout.

Parallel reductions

Let $M \xrightarrow{R} N$ be a one step reduction, where the redex R is contracted. We now extend this notion to a more general case where we contract a *set* of redexes in one step. Let $\mathcal{F} = \{R_1, \dots, R_n\}$ be a set of (possibly nested) redexes in a term M . We write $M \xrightarrow{\mathcal{F}} N$ for the *parallel* contraction of all redexes R_i in M .

Following the presentation given in [Kat90] we can identify three new kinds of reduction:

1. One-degree reduction: If all redexes contracted in a parallel reduction have the same degree.
2. Complete reduction: If the parallel reduction is a one-degree reduction and moreover includes all redexes of a given degree.
3. Leftmost complete reduction: If the parallel reduction is a complete reduction and moreover includes the leftmost redex.

Definition 2.2.14 An optimal reduction of a term M is defined to be the leftmost complete reduction starting at M . $\diamond$

With respect to the graph presentation hinted at above, the theory of optimality says that we should never have two edges in the graph with the same label; they should be shared.

Problems with implementing this theory

We have only given a flavour of the theory here. The basic idea is that redexes with the same label originate from a common source; it is these terms that should be “reduced in parallel” or *shared*. Kathail [Kat90] points out that there are some rather interesting instances of this. For example:

- The redexes being shared may *overlap*:

$$\begin{aligned}
 & - ((\lambda x.(x^a x^b)^c)^d (\lambda x.((\lambda y.x^e)^f b^g)^h)^i)^j \\
 & - ((\lambda x.((\lambda y.x^e)^f b^g)^h)^a \underline{d}i (\lambda x.((\lambda y.x^e)^f b^g)^h)^{b \underline{d}i})^{j \bar{d}c} \\
 & - ((\lambda y.(\lambda x.((\lambda y.x^e)^f b^g)^h)^{ea \underline{d}i b \underline{d}i})^f b^g)^{j \bar{d}ca \bar{d}i h}
 \end{aligned}$$

$(\lambda y.(\lambda x.(\lambda y.x)b))b$ and $(\lambda y.x)b$ originate from the same redex—indeed they have the same degree f —and hence should be shared.

- The redexes being shared may not be the same subexpression. It is this property that gives optimal reduction—the ability to share *partial* applications:

$$\begin{aligned}
& - ((\lambda y.(((y^a a^b)^c (y^d b^e)^f)^g))^h (\lambda x.((\lambda z.z^i)^j x^k)^l)^m)^n \\
& - (((\lambda x.((\lambda z.z^i)^j x^k)^l)^{a\overline{h}m} a^b)^c ((\lambda x.((\lambda z.z^i)^j x^k)^l)^{d\overline{h}m} b^e)^f)^{n\overline{h}g} \\
& - (((\lambda z.z^i)^j a^{ka\overline{h}mb})^{ca\overline{h}ml} ((\lambda x.((\lambda z.z^i)^j x^k)^l)^{d\overline{h}m} b^e)^f)^{n\overline{h}g} \\
& - (((\lambda z.z^i)^j a^{ka\overline{h}mb})^{ca\overline{h}ml} ((\lambda z.z^i)^j b^{kb\overline{h}me})^{f\overline{d\overline{h}ml}})^{n\overline{h}g}
\end{aligned}$$

Ia and *Ib* originate from the same redex—they have the same degree *j*—and hence should be shared.

It is issues like this which make it “hard” to implement Lévy’s theory and is the very reason Lamping introduced the fan-out node to capture this partial sharing.

In this thesis we will make substantial use of Lévy labels to justify many concepts that we use.

- There will be a correspondence between labels on λ -terms and *edges* on graph representations for the λ -calculus. This work has been detailed by Asperti and Laneve [AL93b]. Hence we can reformulate the notion of optimality by stating simply that we never copy an edge in an graph which corresponds to a redex. See *op. cit.* for an excellent treatment of this work.
- In our work on data-flow in Chapter 6 we see that the path computation is exactly the information given by the label, yielding a correspondence between computing a path and reducing a term. Again, these connections can be found in [AL93b] and also [GAL92].

2.2.4 Typed Calculi: PCF

Throughout this thesis we will concentrate on implementing the pure λ -calculus, and investigate extensions for *typed* functional programming languages. In a typed framework we assign types to terms, then have a restricted way in which we can build bigger terms—for example an application MN is only defined if M has type $\alpha \rightarrow \beta$ and N has type α , and we give the term MN the type β . Hence typing restricts the way in which terms can be *plugged together*.

Here we consider a pure functional programming language: PCF (Programming Language for Computable Functions) [Plo77], which is a version of the typed λ -calculus with the addition of constants.

We briefly recall the definition of this language:

Types:

$$\sigma ::= \text{nat} \mid \text{bool} \mid \sigma \rightarrow \tau$$

Terms:

1. $x : \sigma$
2. $c : \sigma$
3. if given $x : \sigma$ we have $M : \tau$ then $\lambda x.M : \sigma \rightarrow \tau$
4. if $M : \sigma \rightarrow \tau$ and $N : \sigma$ then $MN : \tau$

Constants:

1. $n : \text{nat}$
2. $\text{tt}, \text{ff} : \text{bool}$
3. $\text{succ}, \text{pred} : \text{nat} \rightarrow \text{nat}$
4. $\text{iszero} : \text{nat} \rightarrow \text{bool}$
5. for each type σ , $\text{cond}_\sigma : \text{bool} \rightarrow \sigma \rightarrow \sigma \rightarrow \sigma$
6. for each type σ , $Y_\sigma : (\sigma \rightarrow \sigma) \rightarrow \sigma$

Labelled PCF

There is a simple extension of the labelled λ -calculus which will include the constants of PCF. We call this calculus PCF_ℓ .

Definition 2.2.15 (labelled PCF)

To simplify the presentation, which is just an extension of labelled reduction for the λ -calculus, we just show the cases for the PCF constants. Additionally, we will drop types, but of course only well typed expressions are allowed.

we first define:

- $C = \{n, \text{tt}, \text{ff}\}$
- $F = \{\text{succ}, \text{pred}, \text{iszero}, \text{cond}, Y\}$

Let α be a label in $\mathcal{L}$, then the following are labelled PCF terms:

- If $M \in \Lambda'$ is a well typed term, then $M \in \text{PCF}_\ell$
- $c^\alpha \in \text{PCF}_\ell$ for all $c \in C$
- $f^\alpha \in \text{PCF}_\ell$ for all $f \in F$
- If $M, N, P \in \text{PCF}_\ell$ then
 - $\text{succ}^\alpha M \in \text{PCF}_\ell$
 - $\text{pred}^\alpha M \in \text{PCF}_\ell$
 - $\text{iszero}^\alpha M \in \text{PCF}_\ell$
 - $\text{cond}^\alpha M N P \in \text{PCF}_\ell$
 - $Y^\alpha M \in \text{PCF}_\ell$

◇

We define the following extension for the label substitution:

$$\begin{aligned}
 \alpha \cdot c^\beta &= c^{\alpha\beta} \\
 \alpha \cdot f^\beta &= f^{\alpha\beta} \\
 \alpha \cdot (f^\beta M) &= (\alpha \cdot f^\beta)M
 \end{aligned}$$

Labelled PCF delta reduction is now defined by the following clauses which are defined specifically for **cond** and Y , and the rest are named f .

$$\begin{aligned}
 \text{cond}^\alpha \mathbf{tt}^\beta M N &= \alpha\beta \cdot M \\
 \text{cond}^\alpha \mathbf{ff}^\beta M N &= \alpha\beta \cdot N \\
 Y^\alpha M &= (M(Y^\alpha M))^\alpha \\
 f^\alpha c^\beta &= \alpha\beta \cdot (f c)
 \end{aligned}$$

where $(f c)$ are the unlabelled reductions.

Example 2.2.16 We show two simple examples of labelled PCF reduction to show how things work:

1. $(\lambda x. \text{cond } x \mathbf{ff} \mathbf{tt})\mathbf{ff}$

- $((\lambda x. \text{cond}^a x^b \mathbf{ff}^c \mathbf{tt}^d)^e \mathbf{ff}^f)^g$
- $\text{cond}^{g\bar{e}a} \mathbf{ff}^{b\bar{e}f} \mathbf{ff}^c \mathbf{tt}^d$
- $\mathbf{tt}^{g\bar{e}a\bar{b}\bar{e}fd}$

2. $\Omega = Y(\lambda x.x)$

- $Y^a(\lambda x.x^b)^c$
- $((\lambda x.x^b)^c(Y^a(\lambda x.x^b)^c))^a$
- $Y^{a\bar{c}b\bar{c}a}(\lambda x.x^b)^c$
- $\vdots$

We encourage the reader to construct a labelled graph and check that the labels above give a navigation through the graph in the same way as before. $\diamond$

2.2.5 The Linear $\lambda\sigma$ -calculus

In the λ -calculus we are free to copy and discard variables in terms. This can be characterised by the following two λ -terms:

$$\begin{aligned} K &= \lambda x.\lambda y.x \\ W &= \lambda x.\lambda y.xyy \end{aligned}$$

where y is discarded in the first term, and y is copied in the second.

When these terms are applied to arguments KMN and WMN , for some terms M and N , the progress of the substitution through the term will begin. These terms will be copied or discarded by the *meta* operation of substitution.

Since this thesis is about looking more deeply into the notion of reduction in the λ -calculus we feel that the notion of a meta operation is an inadequacy of the theory for our purposes. This motivates the investigation of a variant of the λ -calculus where the progress of a substitution through a term is made explicit. The calculus that we shall use is the Linear $\lambda\sigma$ -calculus, which has been developed by Samson Abramsky and Radha Jagadeesan in studying translations of the λ -calculus into Linear Logic proofs. The significant feature of this calculus is that copying, discarding and substitution become part of the syntax. This leads directly to a decomposed notion of reduction where we can see exactly where substitutions are copied and discarded.

We extend the syntax of the λ -calculus with discarding and copying constructs which are written as $[x = _]M$ and $[x = \langle y, z \rangle]M$ respectively. The first says that x does not occur in M , and the second says that if x occurs twice in M then we call one y and the other z and they are combined by the copying construct. If x occurs more than twice then we can use this rule repeatedly so all occurrences of the variable x get a unique name.

In this calculus variables appear exactly *once* and substitutions are made explicit; hence the name Linear $\lambda\sigma$ -calculus.

Term	Constraint	Free Variables (fv)
x		$\{x\}$
MN	$fv(M) \cap fv(N) = \emptyset$	$fv(M) \cup fv(N)$
$\lambda x.M$	$x \in fv(M)$	$fv(M) \setminus \{x\}$
$M[N/x]$	$x \in fv(M)$ $fv(M) \cap fv(N) = \emptyset$	$(fv(M) \setminus \{x\}) \cup fv(N)$
$[x = _]M$	$x \notin fv(M)$	$fv(M) \cup \{x\}$
$[z = \langle x, y \rangle]M$	$x, y \in fv(M)$ $x \neq y$ $z \notin fv(M)$	$(fv(M) \setminus \{x, y\}) \cup \{z\}$

Figure 2.2: Syntax: Linear $\lambda\sigma$ -calculus

The syntax of $\lambda\sigma$ -terms is given in Figure 2.2. M, N, P are used to range over these terms, and we write $fv(M)$ for the set of variables occurring freely in M . With each term-forming operation there is a syntactic linearity constraint on how it can be applied.

We remark that there is an obvious translation of terms of the λ -calculus into the Linear $\lambda\sigma$ -calculus which does nothing more than count variables to decorate the terms with the explicit copying and discarding operators.

We adopt a variable convention analogous to that in [Bar84] : all bound variables are chosen to be different from free variables. We shall also adopt an abbreviation and write $[\vec{x} = _]$ for

$$[x_1 = _][x_2 = _] \cdots [x_n = _],$$

and similarly for $[\vec{x} = \langle \vec{u}, \vec{v} \rangle]$.

Example 2.2.17

$$\begin{aligned} \mathbf{S} &= \lambda xyz.[z = \langle u, v \rangle](xu)(yv) \\ \mathbf{K} &= \lambda xy.[y = _]x \\ \mathbf{K}_* &= \lambda xy.[x = _]y \end{aligned}$$

◇

Remark 2.2.18 The reader who has tried writing terms in this calculus may have noticed that there are several ways of writing what, at first sight, are the same thing. For example, consider the linear version of the $\mathbf{K}_*$ combinator ($\mathbf{K}_* = \lambda xy.y$); we can write this in two ways:

$$\lambda xy.[x = _]y \text{ and } \lambda x.[x = _]\lambda y.y$$

Operationally, using a leftmost outermost evaluation strategy, the first version takes two arguments, one at a time, then discards the first one. The second version states that after the first argument has been provided, it will discard it, then wait for the second argument. These terms have the same type, but surely the second version is better from a programming point of view—the sooner we can reclaim the data space the better, and hence one should try to place the discarding as close to the outside of the term as possible. Technically, these terms are related via what is known as a commutative conversion; the same term can be given in different ways. Real terms are then those modulo these conversions, but we are forced to make some choices in the syntax. A similar phenomenon also arises in the case of a copying where this time one would want to place the operation deep into the proof so that copying would be performed at the latest possible instance. Our implementations will try to make sure that these special cases are utilised. $\diamond$

The dynamics (or rewriting) are factored into two parts, a structural congruence $\equiv$ and a reduction relation $\rightarrow_\sigma$, following the ideas presented in [Mil90] for the π -calculus. One sees the structural congruence as a mixing rule that puts redexes into juxtaposition so that the reductions can be applied (cf. the Chemical Abstract Machine [BB90]).

The structural congruence is the least congruence $\equiv$ on terms such that:

1. $M \equiv_\alpha N \implies M \equiv N$
2. $M[N/x][P/y] \equiv M[P/y][N/x]$ if $x \notin \text{fv}(P), y \notin \text{fv}(N)$.
3. $M[N/x][P/y] \equiv M[N[P/y]/x]$ if $y \in \text{fv}(N)$.
4. $[x = _]\lambda y.M \equiv \lambda y.[x = _]M$
5. $([x = _]M)[N/y] \equiv [x = _](M[N/y])$ if $x \neq y$
6. $([z = \langle u, v \rangle]M)[N/y] \equiv [z = \langle u, v \rangle](M[N/y])$ if $z \neq y$
7. $[x = _]([y = _]M) \equiv [y = _]([x = _]M)$

The “ β -reductions” are as follows:

1. $(\lambda x.M)N \rightarrow_\sigma M[N/x]$.
2. $x[N/x] \rightarrow_\sigma N$.
3. $([x = _]M)[N/x] \rightarrow_\sigma [\vec{x} = _]M$, where $\vec{x} = \text{fv}(N)$.
4. $([z = \langle x, y \rangle]M)[N/z] \rightarrow_\sigma [\vec{z} = \langle \vec{x}, \vec{y} \rangle](M[N[\vec{x}/\vec{z}]/x][N[\vec{y}/\vec{z}]/y])$, where $\vec{z} = \text{fv}(N)$.

5. $(M[P/x])[Q/y] \rightarrow_\sigma M[P[Q/y]/x]$ if $y \in fv(P)$
6. $(MN)[P/x] \rightarrow_\sigma M(N[P/x])$, if $x \in fv(N)$
7. $(MN)[P/x] \rightarrow_\sigma (M[P/x])N$, if $x \in fv(M)$
8. $(\lambda y.M)[N/x] \rightarrow_\sigma \lambda y.(M[N/x])$, by the variable convention.

and we have the usual η -rule:

$$\lambda x.Mx \rightarrow M$$

Note, that by linearity, there is no side condition to this rule.

These reductions can be applied in any context:

$$\frac{M \rightarrow N}{C[M] \rightarrow C[N]} \text{ context rule}$$

and are performed modulo the structural congruence:

We write $\rightarrow^*$ for the transitive closure, and $=$ for the reflexive closure of $\rightarrow_\sigma$.

$$\frac{M \equiv M' \quad M' \rightarrow N' \quad N' \equiv N}{M \rightarrow N}$$

Example 2.2.19 Here we give a simple example of using the Linear $\lambda\sigma$ -calculus.

$$\begin{aligned}
 \text{SKI} &= ((\lambda xyz.[z = \langle u, v \rangle](xu)(yv))\mathbf{K})\mathbf{I} \\
 &\rightarrow_\sigma ((\lambda yz.[z = \langle u, v \rangle](xu)(yv))[\mathbf{K}/x])\mathbf{I} \\
 &\equiv ((\lambda yz.[z = \langle u, v \rangle](xu)(yv))\mathbf{I})[\mathbf{K}/x] \\
 &\rightarrow_\sigma ((\lambda z.[z = \langle u, v \rangle](xu)(yv))[\mathbf{I}/y])[\mathbf{K}/x] \\
 &\quad \vdots \\
 &\rightarrow_\sigma ((\lambda z.[z = \langle u, v \rangle](xu)(\mathbf{I}v)))[\mathbf{K}/x] \\
 &\quad \vdots
 \end{aligned}$$

◇

It is rule 4. from above that isolates the problem of optimal reduction. If R is a redex in N , then this will be copied; even worse, “potential” (created) redexes will also be copied by the action of copying all the free variables.

Our work will involve particular strategies placed upon this calculus hence we will not address any notion of confluence for this calculus.

2.3 Linear Logic

Linear Logic [Gir87] is a *resource sensitive* logic—resources must be used exactly once. It is the structural rules *weakening* (discarding)

$$\frac{\vdash \Gamma}{\vdash \Gamma, A}$$

and *contraction* (many copies)

$$\frac{\vdash \Gamma, A, A}{\vdash \Gamma, A}$$

that are removed. The logical power is then regained by re-introducing these *structural* rules in a controlled manner as *logical* rules using a modality $!A$ (read “of course A ”) with the intention that a proof of type $!A$ can be used to produce any finite number of copies of a value of type A .

We will call formulae of the form $!A$ *boxed* values, with the intuition that the modality encloses the proof of A which indicates that part of the proof that can be used in a non-linear way. We will make this more precise below.

There are three important consequences of this logic with respect to the logical rules that we need to introduce.

- We need logical rules to copy, discard and *open* values of type $!A$. These rules are given below under the names Contraction, Weakening and Dereliction. In addition we need a rule to introduce a type $!A$ which is restricted by the fact that since we allow non-linear use of this type, we must also insist that the context is built up of types that can be used in a non-linear way, *i.e.* linearity is propagated through a proof.
- The elimination of the structural rules immediately gives a decomposition of the conjunction and disjunction into *Multiplicative* and *Additive* connectives.

	Multiplicative	Additive
Conjunction	$\otimes$ (tensor)	$\&$ (with)
Disjunction	$\wp$ (par)	$\oplus$ (sum)

We refer the reader to the source of Linear Logic [Gir87] for further details of these issues.

- We get a constructive classical logic with de-Morgan dualities with the *linear negation* $A^\perp$ (read “ A perp”). Linear negation is then *definitionally* extended to formulae of Linear Logic as follows:

$$\begin{aligned}
A^{\perp\perp} &= A \\
(A \otimes B)^\perp &= A^\perp \wp B^\perp \\
(!A)^\perp &= ?A^\perp
\end{aligned}$$

With ? (read “why not”) being the dual of the of course !.

The *linear implication* $A \multimap B$ is then defined as an abbreviation for $A^\perp \wp B$ (cf. $\neg A \vee B$).

We are going to use Linear Logic proof structures as a representation of the λ -calculus. The fragment of the logic that we need is exactly the Multiplicative Exponential fragment, which we write as MELL. Linear Logic proof structures provide the right kind of structure that we are seeking for our implementations since it makes clear the usage of proofs (programs) during the cut elimination (normalisation) process.

There are many presentations of Linear Logic that one can find in the literature. Here we present the one-sided sequent calculus version of Classical Linear Logic (CLL). This is made possible by the following fact:

$$\Gamma \vdash \Delta \iff \vdash \Gamma^\perp, \Delta$$

We present the logical system by distinguishing between the various groups of the logic. The presentation is given in Figure 2.3.

The rule for promotion deserves special mention:

$$\frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A}$$

First, the notation $?\Gamma$ refers to a context Γ where all the elements are of the form $?A_i$. Hence the promotion rule can only be applied to a context where all the assumptions are non-linear values. Secondly, with respect to the terminology of a *box* we should regard *the entire proof* above the promotion rule as being boxed. There are two additional pieces of terminology that we use frequently. The formulae that come out of the box do so at two distinct places. The formula $!A$ is said to be at the *principal* or *main* door of the box; and the formulae $?\Gamma$ are all said to be at the *auxiliary* or *side* doors of the box.

Identity Group	
$\frac{}{\vdash A^\perp, A}$ (Axiom)	$\frac{\vdash \Gamma, A^\perp \quad \vdash \Delta, A}{\vdash \Gamma, \Delta}$ (Cut)
Structural Rule	
$\frac{\vdash \Gamma, A, B, \Delta}{\vdash \Gamma, B, A, \Delta}$ (Exchange)	
Logical Rules: Multiplicatives	
$\frac{\vdash \Gamma, A \quad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B}$ (Tensor)	$\frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \wp B}$ (Par)
Logical Rules: Exponentials	
$\frac{\vdash \Gamma, A}{\vdash \Gamma, ?A}$ (Dereliction)	
$\frac{\vdash \Gamma}{\vdash \Gamma, ?A}$ (Weakening)	$\frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A}$ (Contraction)
$\frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A}$ (Promotion)	

Figure 2.3: Linear Logic

Proof Rule	Operation	Constraint	Free Names (fn)
Axiom	$I_{x,y}$		$\{x, y\}$
Cut	$P \cdot_x Q$	$fn(P) \cap fn(Q) = \{x\}$	$(fn(P) \cup fn(Q)) \setminus \{x\}$
Times	$\otimes_z^{x,y}(P, Q)$	$x \in fn(P), y \in fn(Q)$ $fn(P) \cap fn(Q) = \emptyset$ $z \notin fn(P) \cup fn(Q)$	$(fn(P) \cup fn(Q) \setminus \{x, y\}) \cup \{z\}$
Par	$\wp_z^{x,y}(P)$	$x, y \in fn(P)$ $x \neq y$ $z \notin fn(P)$	$(fn(P) \setminus \{x, y\}) \cup \{z\}$
Dereliction	$D_z^x(P)$	$x \in fn(P), z \notin fn(P)$	$(fn(P) \setminus \{x\}) \cup \{z\}$
Weakening	$W_z(P)$	$z \notin fn(P)$	$fn(P) \cup \{z\}$
Contraction	$C_z^{x,y}(P)$	$x, y \in fn(P)$ $x \neq y$ $z \notin fn(P)$	$(fn(P) \setminus \{x, y\}) \cup \{z\}$
Of course	$!_z^x(P)$	$x \in fn(P), z \notin fn(P)$	$(fn(P) \setminus \{x\}) \cup \{z\}$

Figure 2.4: Syntax: Linear Realisability Algebras

2.4 Linear Realisability Algebras

Linear Realisability Algebras (LRA's) were introduced in Abramsky's Proofs as Processes lecture [Abr91] and developed in [AJ94] as a notion of an abstract algebra for Linear Logic proofs (*cf.* combinatory algebras for the λ -calculus). Therefore we can use LRA's as a representation of a proof in Linear Logic. We will follow the presentation given in *op. cit.*; restricting the presentation of the algebra to the fragment of the logic that we are interested in.

We first introduce the syntax of terms which we show in Figure 2.4, which will be used as *realisers* for sequent proofs in CLL. Let P, Q, R to range over these terms, and write $fn(P)$ for the set of names occurring freely in P . With each term-forming operation we give a linearity constraint on how it can be applied, and specify its free names.

As with the Linear $\lambda\sigma$ -calculus, there is an evident notion of *renaming* $P[x/y]$ satisfying the following:

$$\begin{aligned}
 I_{x,y}[z/x] &= I_{z,y} \text{ and similarly for } y \\
 (P \cdot_x Q)[z/x] &= P[z/x] \cdot_z Q[z/x]
 \end{aligned}$$

and of α -conversion $P \equiv_\alpha Q$. We will also adopt a similar abbreviation and write

$W_{\vec{x}}(P)$ for

$$W_{x_1}(\cdots W_{x_n}(P) \cdots)$$

and similarly for $C_z^{\vec{x}, \vec{y}}$.

Definition 2.4.1 (Dynamics)

We split the dynamics as before into two parts—a structural congruence and a reduction relation.

Structural congruence:

1. $P \equiv_{\alpha} Q \implies P \equiv Q$
2. $P \cdot_x Q \equiv Q \cdot_x P$
3. $(P \cdot_x Q) \cdot_y R \equiv P \cdot_x (Q \cdot_y R)$
4. $\wp_z^{x,y}(P) \cdot_v Q \equiv \wp_z^{x,y}(P \cdot_v Q)$ if $v \in fn(P)$
5. $\otimes_z^{x,y}(P, R) \cdot_v Q \equiv \otimes_z^{x,y}(P \cdot_v Q, R)$ if $v \in fn(P)$, and similarly for $v \in fn(R)$.
6. $C_z^{x,y}(P) \cdot_v Q \equiv C_z^{x,y}(P \cdot_v Q)$ if $v \in fn(P)$
7. $W_z(P) \cdot_v Q \equiv W_z(P \cdot_v Q)$ if $v \in fn(P)$
8. $D_z^x(P) \cdot_v Q \equiv D_z^x(P \cdot_v Q)$ if $v \in fn(P)$

The justification of these equations comes from the connection with Proof Nets which we will discuss at the end of this chapter.

Reductions:

The reductions are decomposed into two parts—“ β ”-reductions and the dual “ η ”-reductions:

- β -reductions:

1. $P \cdot_x I_{x,y} \rightarrow P[y/x]$
2. $\wp_z^{x,y}(P) \cdot_z \otimes_z^{u,v}(Q, R) \rightarrow P \cdot_x Q[x/u] \cdot_y R[y/v]$
3. $D_z^x(P) \cdot_z !_z^y(Q) \rightarrow P \cdot_x Q[x/y]$
4. $W_z(P) \cdot_z !_z^y(Q) \rightarrow W_{\vec{x}}(P)$, where $\vec{x} = fn(Q)$
5. $C_z^{x,y}(P) \cdot_z !_z^u(Q) \rightarrow C_{\vec{c}}^{\vec{a}, \vec{b}}(P \cdot_x !(Q[\vec{a}/\vec{c}]) \cdot_y !(Q[\vec{b}/\vec{c}]))$ where $\vec{c} = fn(Q)$
6. $!_y^x(P) \cdot_v !_v^u(Q) \rightarrow !_y^x(P \cdot_v !_v^u(Q))$ if $v \in fn(P)$

- η -reductions

$$1. \wp_v^{x,a} C[(\otimes_s^{y,b}(I_{x,y}, I_{a,b}))] \rightarrow C[I_{v,s}]$$

$$2. !_v^y C[(D_z^x(I_{x,y}))] \rightarrow C[I_{z,v}]$$

(The author has found it instructive to draw these η rules as proof net reductions which brings out the symmetry with the β reductions.)

◇

Again, the justification of these equations will be discussed in the context of Proof Nets at the end of this chapter.

As before, the reductions can be applied in any context:

$$\frac{P \rightarrow Q}{C[P] \rightarrow C[Q]} \text{ context rule}$$

and are performed modulo the structural congruence:

$$\frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q}$$

Again, the context rule can be restricted to give different evaluation orders.

Definition 2.4.2 (Normal forms)

Let P and Q be arbitrary LRA's. The set of normal forms N of the theory is defined as:

$$\begin{aligned} N ::= & I_{x,y} \mid !P \\ & \mid !P \cdot_x !Q \text{ if } x \notin fn(P) \\ & \mid \wp(N) \mid \otimes(N, N') \\ & \mid D(N) \end{aligned}$$

◇

2.4.1 Assignments of LRA's to Classical Linear Logic Proofs

Here we show the assignment of Linear Realisability Algebras to Linear Logic proofs (cf. the assignment of the λ -calculus to Intuitionistic Logic proofs). Hence we are now dealing with *typed* LRA's. We write $P \vdash \Gamma$ for P is a program (or realiser) for Γ . The presentation is given in Figure 2.5. Note particularly that there is no term construction for the structural rule.

We can now justify the rewriting equations for Linear Realisability Algebras (Definition 2.4.1).

Identity Group $\frac{}{I_{x,y} \vdash x : A^\perp, y : A} \text{ (Axiom)}$ $\frac{P \vdash \Gamma, x : A^\perp \quad Q \vdash \Delta, x : A}{P \cdot_x Q \vdash \Gamma, \Delta} \text{ (Cut)}$
Structural Rule $\frac{P \vdash \Gamma, A, B, \Delta}{P \vdash \Gamma, B, A, \Delta} \text{ (Exchange)}$
Logical Rules: Multiplicatives $\frac{P \vdash \Gamma, x : A \quad Q \vdash \Delta, y : B}{\otimes_z^{x,y}(P, Q) \vdash \Gamma, \Delta, z : A \otimes B} \text{ (Tensor)}$ $\frac{P \vdash \Gamma, x : A, y : B}{\wp_z^{x,y}(P) \vdash \Gamma, z : A \wp B} \text{ (Par)}$
Logical Rules: Exponentials $\frac{P \vdash \Gamma, x : A}{D_y^x(P) \vdash \Gamma, y : ?A} \text{ (Dereliction)}$ $\frac{P \vdash \Gamma}{W_z(P) \vdash \Gamma, z : ?A} \text{ (Weakening)}$ $\frac{P \vdash \Gamma, x : ?A, y : ?A}{C_z^{x,y}(P) \vdash \Gamma, z : ?A} \text{ (Contraction)}$ $\frac{P \vdash ?\Gamma, x : A}{!_z^x(P) \vdash ?\Gamma, z : !A} \text{ (Promotion)}$

Figure 2.5: Assignment of LRA's to Linear Logic proofs

Proposition 2.4.3

- If $P \vdash \Gamma$ and $P \equiv Q$ then $Q \vdash \Gamma$.
- If $P \vdash \Gamma$ and $P \rightarrow Q$ then $Q \vdash \Gamma$.

◇

PROOF: Straightforward induction over the length of the proof.

□

2.5 LRA's and Proof Nets

In this section we relate LRA's with *proof nets* [Gir87] to justify the equational theory presented.

Proof nets provide a very natural syntax for Linear Logic proofs. The principal aim is to provide a *natural deduction* presentation of a seemingly sequent based logic. These networks are a two-dimensional syntax which free us from many of the ordering properties of a sequent proof. The salient feature of proof nets is that they provide a natural deduction system for a multi-conclusion logic. We refer the reader to the source (*op. cit.*) for a presentation of proof nets.

In this thesis we will talk about Linear Logic proof structures using LRA's, which can be seen as an attempt to *linearise* the two-dimensional nets. This collapse in dimensions however has its problems in that we are forced to make some choices again on the order of use of rules. This is exactly why we needed to introduce a structural congruence on our syntax.

There is an evident notion of translating an LRA proof into a proof net—we simply draw the elements of the term in two-dimensional space and draw a line between connecting nodes as indicated by the names of the channels. The identity $I_{x,y}$ is translated into a piece of wire—a connecting edge, and a cut $P \cdot_x Q$ is represented by connecting the free name x of the separate nets together.

We are now in a position to state the following

Proposition 2.5.1 Let T be the translation of LRA's to proof nets and P and Q be LRA's such that $P \equiv Q$, then $T(P) = T(Q)$. So the structural congruence on LRA's corresponds to equality of nets.

◇

PROOF: Straightforward case analysis over why $P \equiv Q$.

□

Moreover, we have a tight correspondence between reductions in LRA's and reductions in proof nets.

Proposition 2.5.2 Let P and Q be LRA's such that $P \rightarrow Q$ is a valid one-step β -reduction. Then $T(P)$ reduces in one step to $T(Q)$. $\diamond$

PROOF: Straightforward analysis of the reduction relation $\rightarrow$. $\square$

We will always consider those proof nets arising from a given LRA. Hence a proof net $T(P)$ is correct just when there is a proof $P \vdash \Gamma$.

2.6 Discussion

In this chapter we have presented a sequence of calculi that we claim gives rise to a series of refinements of the λ -calculus. Our aim is to use Linear Logic proof structures to represent the λ -calculus and we used the Linear $\lambda\sigma$ -calculus as an intermediate calculus which is not so far removed from the λ -calculus, but makes our translations simpler and easier to understand.

In the next chapter we will look at translating the λ -calculus into Linear Logic proof structures and investigate what natural orders of evaluation are induced on the λ -terms. We will then go on, in Chapters 5 and 6, to extracting the “answer” of a computation in these structures, using ideas from the *Geometry of Interaction* interpretation of Linear Logic.

Chapter 3

λ -calculus in Proof Structures

This chapter sets up the starting point of all our implementations by embedding the λ -calculus into Linear Logic proof structures.

3.1 Introduction

It is our thesis that implementing the λ -calculus through “the Linear Logic looking glass” gives new techniques at a very fine level amenable to simple code generation algorithms. This chapter sets out the foundations for this by defining several translations of the λ -calculus into Linear Logic proof structures. We will look at three translations, each of which induces a different notion of reduction back in the λ -calculus.

Much of the work of this chapter is not new, and two of the translations given have been presented in many pieces of work. We must therefore justify our re-presentation.

- First, we present the translations in our notation which we shall use throughout the rest of this thesis. In particular, we will show the translations from the λ -calculus into Linear Realisability Algebras.
- We state and prove several properties of these translations which are generally taken for granted.
- We try to understand and clarify which translation is the best to use for an implementation of a functional programming language by examining the number of reduction steps and the kind of reductions induced.
- Finally, we will look at how we can extend these structures with constants so that we can implement a PCF like language.

As discussed in the previous chapter we will use the Linear $\lambda\sigma$ -calculus as a starting point for our translations—since this provides a smoother presentation without the need to perform syntactical analyses on the terms such as variable counting which is made explicit in our calculus.

3.2 Translations

In the literature we find two widely used translations of the λ -calculus into Linear Logic proof structures. We recall these, and set them out in our notation together with a third.

The translations are specified at the *logical level*. For each we state how an intuitionistic judgement $\Gamma \vdash A$ is translated into the one-sided presentation of Linear Logic, so we are giving a translation of intuitionistic proofs into Linear Logic proofs. The names of the various translations will be given by the way the function space is coded; following the literature.

1. The $!A \multimap B$ translation takes the intuitionistic judgement $\Gamma \vdash A$ into Linear Logic as $\vdash ?[\Gamma]^\perp, [A]$ where:

$$\begin{aligned} [A] &= A \text{ (} A \text{ atomic)} \\ [A \rightarrow B] &= ![A] \multimap [B] \end{aligned}$$

2. The $!(A \multimap B)$ translation takes the intuitionistic judgement $\Gamma \vdash A$ into Linear Logic as $\vdash [\Gamma]^\perp, [A]$ where:

$$\begin{aligned} [A] &= !A \text{ (} A \text{ atomic)} \\ [A \rightarrow B] &= !([A] \multimap [B]) \end{aligned}$$

3. The $!A \multimap !B$ translation takes the intuitionistic judgement $\Gamma \vdash A$ into Linear Logic as $\vdash ?[\Gamma]^\perp, ![A]$ where:

$$\begin{aligned} [A] &= A \text{ (} A \text{ atomic)} \\ [A \rightarrow B] &= ![A] \multimap ![B] \end{aligned}$$

The codings of intuitionistic proofs into Linear Logic proofs gives rise to a very beautiful aspect of the Curry-Howard isomorphism. That is, the translation at the logical level induces a translation at the term level. Hence we get a translation from the λ -calculus into Linear Realisability algebras or proof nets. We present these using both notions of Linear Logic proof in parallel. Proof nets are our “official” notation, but LRA’s are easier to typeset!

The presentation we give here is for the *untyped* λ -calculus. It is a straightforward exercise to derive the typed versions, and we briefly mention this when we look at PCF. In particular, we shall write our translations as solving fix-points of various recursive type equations:

- $D = !D \multimap D$
- $D = !(D \multimap D)$
- $D = !D \multimap !D$

In the following we will present each of these translations in turn, then go on to show various properties of the translations.

Notation

The one-sided presentation of Linear Logic in the sequent calculus given in Figure 2.3 hides the notion of *input* and *output* that is inherent in the λ -calculus. The one sided sequents that we will write should have an intended orientation following that of intuitionistic logic. When we write $\Gamma \vdash A$ (input Γ , output A) as $\vdash \Gamma^\perp, A$ it becomes very difficult to distinguish inputs from outputs. We will adopt a notational convention that the (unique) output is always the rightmost formula of the sequent, and to make this even more explicit, we will adopt the following notation:

$$\vdash \Gamma ; A$$

where “;” is really a “,” but makes the notation human readable.

The translations that we give below can be understood as defining a proof structure for each term construction in the Linear $\lambda\sigma$ -calculus. An alternative way at viewing this is thinking of the Linear $\lambda\sigma$ -calculus in a *typed* setting taking the universal type $U = U \rightarrow U$. Hence we can write $\lambda\sigma$ judgements $\Gamma \vdash t : U$ in a similar way to the simply typed λ -calculus. We will define translations of these judgements to the corresponding version in Linear Logic proof structures:

1. $T[[t]] \vdash ?[\Gamma]^\perp ; [U]$
 where we define $[U \rightarrow U] = !D \multimap D$
2. $T[[t]] \vdash [\Gamma]^\perp ; [U]$
 where we define $[U \rightarrow U] = !(D \multimap D)$
3. $T[[t]] \vdash ?[\Gamma]^\perp ; ![\Gamma]$

were we define $[U \rightarrow U] = !D \multimap !D$

Note that in these *recursive type* translations we do not have a translation of base types; all types are functional.

3.2.1 $D = !D \multimap D$

Our first translation is based on the recursive type equations:

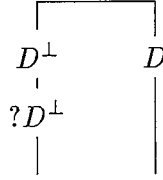
$$\begin{aligned} D &= !D \multimap D = ?D^\perp \wp D \\ D^\perp &= !D \otimes D^\perp \end{aligned}$$

which has been used by Danos and Regnier [DR93], and Asperti and Laneve [AL91, AL93a] in their work on Virtual reduction and Optimal reduction in the λ -calculus respectively. This is the “standard” translation and its origins are Linear Logic, in the sense that the intuitionistic arrow $A \rightarrow B$ is decomposed into $!A \multimap B$, i.e. linear implication and repeated use of argument.

Definition 3.2.1 ($T_{!D \multimap D}$)

The translation function $T[\![\cdot]\!]$ is defined inductively over the structure of $\lambda\sigma$ -terms as follows:

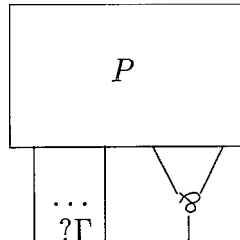
Variable: We define $T[\![x]\!]$ as the following proof structure:



The sequent presentation of this proof structure is given by:

$$\frac{I_{u,v} \vdash u : D^\perp ; v : D}{D_x^u(I_{u,v}) \vdash x : ?D^\perp ; v : D}$$

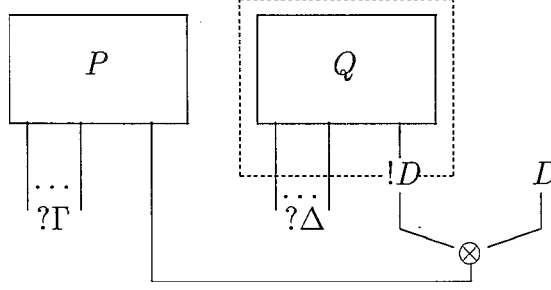
Abstraction: Let $T[\![M]\!] = P$, then we define $T[\![\lambda x.M]\!]$ as the following proof structure:



which corresponds to the following sequent proof:

$$\frac{P \vdash ?\Gamma, x : ?D^\perp ; y : D}{\wp_z^{x,y}(P) \vdash ?\Gamma ; z : ?D^\perp \wp D = D}$$

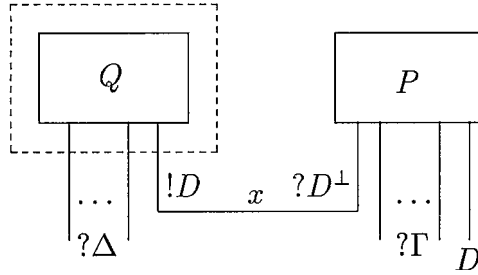
Application: Let $T[M] = P$ and $T[N] = Q$, then we define $T[MN]$ as:



We have used a *dashed line* to represent the *!* box. The terminology of a box, side door and main doors now becomes evident.

$$\frac{\frac{Q \vdash ?\Delta ; u : D}{!_v^u(Q) \vdash ?\Delta ; v : !D} \quad I_{x,y} \vdash x : D^\perp ; y : D}{\frac{P \vdash ?\Gamma ; w : D \quad \otimes_w^{v,x}(!_v^u(Q), I_{x,y}) \vdash ?\Delta, w : !D \otimes D^\perp = D^\perp ; y : D}{P \cdot w \otimes_w^{v,x}(!_v^u(Q), I_{x,y}) \vdash ?\Gamma, ?\Delta ; y : D}}$$

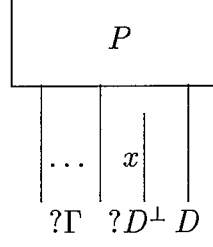
Substitution: Let $T[M] = P$ and $T[N] = Q$, then we define the substitution $T[M[N/x]]$ as:



which is the representation of the following sequent proof:

$$\frac{P \vdash ?\Gamma, x : ?D^\perp ; y : D \quad \frac{Q \vdash ?\Delta ; z : D}{!_x^z(Q) \vdash ?\Delta ; x : !D}}{P \cdot_x !_x^z(Q) \vdash ?\Gamma, ?\Delta ; y : D}$$

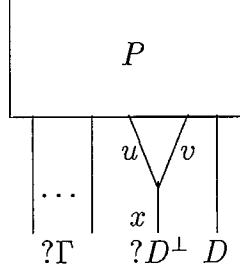
Weakening: Let $T[[M]] = P$, then we define $T[[x = _]M]]$ as:



which is the representation of the following sequent proof:

$$\frac{P \vdash ?\Gamma ; y : D}{W_x(P) \vdash ?\Gamma, x : ?D^\perp ; y : D}$$

Contraction: Let $T[[M]] = P$, then we define $T[[x = \langle u, v \rangle]M]]$ as:



which is the representation of the following sequent proof:

$$\frac{P \vdash \Gamma, u : ?D^\perp, v : ?D^\perp ; y : D}{C_x^{u,v}(P) \vdash \Gamma, x : ?D^\perp ; y : D}$$

◇

Example 3.2.2 To give some idea of how things work, we show a very simple example of a reduction of a λ -term in Linear Realisability Algebras.

$$\begin{aligned} T[[\lambda x.x)(\lambda x.x]] &= \wp(D(I)) \cdot \otimes(!(\wp(D(I))), I) \\ &\rightarrow D(I) \cdot !(\wp(D(I))) \cdot I \\ &\rightarrow I \cdot \wp(D(I)) \cdot I \\ &\rightarrow \wp(D(I)) \\ &= T[[\lambda x.x]] \end{aligned}$$

◇

3.2.2 $D = !(D \multimap D)$

The second translation is based on the recursive type equation

$$\begin{aligned} D &= !(D \multimap D) = !(D^\perp \wp D) \\ D^\perp &= ?(D \otimes D^\perp) \end{aligned}$$

and has been used by Gonthier, Abadi and Lévy [GAL92] in the refinement of Lamping's implementation of Optimal Reduction. This translation is somewhat dual to the previous one in that we promote the function rather than the argument.

Definition 3.2.3 ($T_{!(D \multimap D)}$)

Variables: We define $T[x]$ as:

$$\begin{array}{c} \overline{} \\ D^\perp \quad D \\ | \quad | \end{array}$$

The sequent presentation of this proof structure is given by:

$$\overline{I_{x,y} \vdash x : D^\perp ; y : D}$$

Abstraction: Let $T[M] = P$, then we define $T[\lambda x.M]$ as:

$$\frac{\frac{P \vdash \Gamma, x : D^\perp ; y : D}{\wp_z^{x,y}(P) \vdash \Gamma ; z : D^\perp \wp D}}{!_u^z(\wp_z^{x,y}(P)) \vdash \Gamma ; u : !(D^\perp \wp D) = D}$$

Note that since the context Γ is built up of formulæ of type $D^\perp = ?(D \otimes D^\perp)$ the promotion rule is valid.

Application: Let $T[M] = P$ and $T[N] = Q$, then we define $T[MN]$ as:

$$\frac{\frac{\frac{Q \vdash \Delta, u : D \quad I_{x,y} \vdash x : D^\perp ; y : D}{\otimes_z^{u,x}(Q, I_{x,y}) \vdash \Delta, z : D \otimes D^\perp ; y : D}}{D_v^z(\otimes_z^{u,x}(Q, I_{x,y})) \vdash \Delta, v : ?(D \otimes D^\perp) = D^\perp ; y : D}}{P \cdot_v D_v^z(\otimes_z^{u,x}(Q, I_{x,y})) \vdash \Gamma, \Delta ; y : D}$$

Substitution: Let $T[M] = P$ and $T[N] = Q$, then we define the substitution $T[M[N/x]]$ as:

$$\frac{P \vdash \Gamma, x : D^\perp ; y : D \quad Q \vdash \Delta ; x : D}{P \cdot_x Q \vdash \Gamma, \Delta ; y : D}$$

Weakening: Let $T[M] = P$, then we define $T[[x = _]M]$ as:

$$\frac{P \vdash \Gamma ; y : D}{W_x(P) \vdash \Gamma, x : D^\perp ; y : D}$$

Note that this works since $D^\perp = ?(D \otimes D^\perp)$, and similarly for the contraction rule below.

Contraction: Let $T\llbracket M \rrbracket = P$, then we define $T\llbracket [x = \langle u, v \rangle]M \rrbracket$ as:

$$\frac{P \vdash \Gamma, u : D^\perp, v : D^\perp ; y : D}{C_x^{u,v}(P) \vdash \Gamma, x : D^\perp ; y : D}$$

◇

3.2.3 $D = !D \multimap !D$

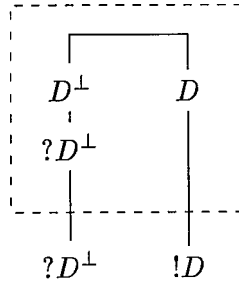
Our final translation is based on the recursive type equation

$$\begin{aligned} D &= !D \multimap !D = ?D^\perp \wp !D \\ D^\perp &= !D \otimes ?D^\perp \end{aligned}$$

This translation has been studied by Lafont, but to our knowledge has not been documented.

Definition 3.2.4 ($T_{!D \multimap !D}$)

Variables: We define $T\llbracket x \rrbracket$ as:



which is the representation of the following sequent proof:

$$\frac{\frac{I_{u,v} \vdash u : D^\perp ; v : D}{D_x^u(I_{u,v}) \vdash x : ?D^\perp ; v : D}}{!_y^v(D_x^u(I_{u,v})) \vdash x : ?D^\perp ; y : !D}$$

Abstraction: Let $T\llbracket M \rrbracket = P$, then we define $T\llbracket \lambda x.M \rrbracket$ as:

$$\frac{\frac{P \vdash ?\Gamma, x : ?D^\perp ; y : !D}{\mathcal{R}_z^{x,y}(P) \vdash ?\Gamma ; z : ?D^\perp \mathcal{R}!D = D}}{!_u^z(\mathcal{R}_z^{x,y}(P)) \vdash ?\Gamma ; u : !D}$$

Application: Let $T\llbracket M \rrbracket = P$ and $T\llbracket N \rrbracket = Q$, then we define $T\llbracket MN \rrbracket$ as:

$$\frac{\frac{\frac{I_{x,y} \vdash x : D^\perp ; y : D}{D_z^x(I_{x,y}) \vdash z : ?D^\perp ; y : D}}{Q \vdash ?\Delta ; u : !D \quad !_v^y(D_z^x(I_{x,y})) \vdash z : ?D^\perp ; v : !D}}{\frac{\otimes_{w'}^{u,z}(Q, !_v^y(D_z^x(I_{x,y}))) \vdash ?\Delta, w' : !D \otimes ?D^\perp = D^\perp ; v : !D}{\frac{P \vdash ?\Gamma ; w : !D \quad D_w^{w'}(\otimes_{w'}^{u,z}(Q, !_v^y(D_z^x(I_{x,y})))) \vdash ?\Delta, w : ?D^\perp ; v : !D}{P \cdot_w D_w^{w'}(\otimes_{w'}^{u,z}(Q, !_v^y(D_z^x(I_{x,y})))) \vdash ?\Gamma, ?\Delta ; v : !D}}}$$

Substitution: Let $T\llbracket M \rrbracket = P$ and $T\llbracket N \rrbracket = Q$, then we define the substitution $T\llbracket M[N/x] \rrbracket$ as:

$$\frac{P \vdash \Gamma, x : ?D^\perp ; z : !D \quad Q \vdash ?\Delta ; x : !D}{P \cdot_x Q \vdash \Gamma, ?\Delta ; z : !D}$$

Weakening: Let $T\llbracket M \rrbracket = P$, then we define $T\llbracket [x = _]M \rrbracket$ as:

$$\frac{P \vdash ?\Gamma ; y : !D}{W_x(P) \vdash ?\Gamma, x : ?D^\perp ; y : !D}$$

Contraction: Let $T\llbracket M \rrbracket = P$, then we define $T\llbracket [x = \langle u, v \rangle]M \rrbracket$ as:

$$\frac{P \vdash ?\Gamma, u : ?D^\perp, v : ?D^\perp ; y : !D}{C_x^{u,v}(P) \vdash ?\Gamma, x : ?D^\perp ; y : !D}$$

◇

3.2.4 Summary

In Tables 3.1, 3.2 and 3.3 we summarise the translation for the LRA's which we will use repeatedly throughout the thesis. For each translation, we assume $T\llbracket M \rrbracket = P$ and $T\llbracket N \rrbracket = Q$. To give the orientation of the LRA (proof net) we specify the output/result/principal-port of each term construction, which we shall write as *pp*.

We also use the names of the variables in the Linear $\lambda\sigma$ -calculus to give the names of the channels in the LRA's where possible.

$\lambda\sigma$ -term	Translation	pp
x	$D_x^u(I_{u,v})$	v
MN	$P \cdot_{pp(P)} \otimes_{pp(P)}^{v,x} (!_v^{pp(Q)}(Q), I_{x,y})$	y
$\lambda x.M$	$\wp_z^{x,pp(P)}(P)$	z
$M[N/x]$	$P \cdot_x !_x^{pp(Q)}(Q)$	$pp(P)$
$[x = _]M$	$W_x(P)$	$pp(P)$
$[x = \langle u, v \rangle]M$	$C_x^{u,v}(P)$	$pp(P)$

Table 3.1: $!D \multimap D$ translation

$\lambda\sigma$ -term	Translation	pp
x	$I_{x,y}$	y
MN	$P \cdot_{pp(P)} D_{pp(P)}^z (\otimes_z^{pp(Q),x} (Q, I_{x,y}))$	y
$\lambda x.M$	$!_u^z (\wp_z^{x,pp(P)}(P))$	u
$M[N/x]$	$P \cdot_x Q$	$pp(P)$
$[x = _]M$	$W_x(P)$	$pp(P)$
$[x = \langle u, v \rangle]M$	$C_x^{u,v}(P)$	$pp(P)$

Table 3.2: $!(D \multimap D)$ translation

$\lambda\sigma$ -term	Translation	pp
x	$!_y^v (D_x^u(I_{u,v}))$	y
MN	$P \cdot_w D_w^{w'} (\otimes_{w'}^{pp(Q),z} (Q, !_v^y (D_z^x (I_{x,y}))))$	v
$\lambda x.M$	$!_u^z (\wp_z^{x,pp(P)}(P))$	u
$M[N/x]$	$P \cdot_x Q$	$pp(P)$
$[x = _]M$	$W_x(P)$	$pp(P)$
$[x = \langle u, v \rangle]M$	$C_x^{u,v}(P)$	$pp(P)$

Table 3.3: $!D \multimap !D$ translation

Proposition 3.2.5 $fn(P) \setminus pp(P) = fv(M)$. $\diamond$

We point out one feature of all these translations to give the reader some insight into the differences between them and explain how they work. For all the translations we have shown the multiplicative part is the same— $\otimes$ for the compilation of application and $\wp$ for the translation of abstraction. The differences are therefore in where exponentials are placed.

- For the $D = !D \multimap D$ translation we promote the *argument*, and derelict all variables.
- For the $D = !(D \multimap D)$ translation we promote the *function*, and the action of application is to derelict the function so that it can be applied.
- For the $D = !D \multimap !D$ translation, the exponentials are placed in exactly the same place as for the $D = !(D \multimap D)$, but in addition all the variables are boxed derelictions.

By the η rules that we have given for LRA's we see that the last two translations are equivalent. After this chapter we will only consider the first two translations.

3.3 Properties of the translations

In this Section we take a closer look at the translations and examine what kind of evaluation scheme is induced on the λ -terms. In the literature we find the following:

- $D = !D \multimap D$ gives call-by-name
- $D = !(D \multimap D)$ gives call-by-value

To partially justify these results, we show for each translation that the corresponding reduction in the Linear $\lambda\sigma$ -calculus is valid.

Proposition 3.3.1 For the $D = !D \multimap D$ translation we have:

1. $\beta : (\lambda x.M)N \rightarrow M[N/x]$ is valid.
2. $\eta : \lambda x.Mx \rightarrow M$ is valid.

$\diamond$

PROOF:

1. We show that $T[(\lambda x.M)N] \rightarrow^* T[M[N/x]]$ which is straightforward from the definitions (we omit the names of the channels to aid readability):

$$\begin{aligned}
T\llbracket(\lambda x.M)N\rrbracket &= \wp(P) \cdot \otimes(!\langle Q \rangle, I) \\
&\rightarrow P \cdot \langle Q \rangle \cdot I \\
&\rightarrow P \cdot \langle Q \rangle \\
&= T\llbracket M[N/x] \rrbracket
\end{aligned}$$

It remains now to show that substitution $M[N/x]$ is simulated. We show this by induction on the structure of M . Let $T\llbracket A \rrbracket = P$, $T\llbracket B \rrbracket = Q$ and $T\llbracket N \rrbracket = R$.

There are seven cases, we just show four:

- (a) If $M \equiv x$, then $T\llbracket x[N/x] \rrbracket = D(I) \cdot \langle R \rangle \rightarrow I \cdot R \rightarrow R = T\llbracket N \rrbracket$.
- (b) If $M \equiv \lambda y.A$, then

$$\begin{aligned}
T\llbracket(\lambda y.A)[N/x]\rrbracket &= \wp(P) \cdot \langle R \rangle \\
&\equiv \wp(P \cdot \langle R \rangle) \\
&= T\llbracket \lambda y.A[N/x] \rrbracket
\end{aligned}$$

- (c) If $M \equiv AB$, then, assuming $x \in \text{fv}(B)$,

$$\begin{aligned}
T\llbracket(AB)[N/x]\rrbracket &= (P \cdot \otimes(!\langle Q \rangle, I)) \cdot \langle R \rangle \\
&\equiv P \cdot \otimes(!\langle Q \rangle \cdot \langle R \rangle, I) \\
&= P \cdot \otimes(!\langle Q \cdot \langle R \rangle \rangle, I) \\
&= T\llbracket A(B[N/x]) \rrbracket
\end{aligned}$$

- (d) If $M \equiv [x = _]A$ then,

$$\begin{aligned}
T\llbracket([x = _]A)[N/x]\rrbracket &= W_x(P) \cdot \langle R \rangle \\
&\rightarrow W_{\vec{x}}(P) \\
&= T\llbracket([\vec{x} = _]A) \rrbracket
\end{aligned}$$

2. To show that η is valid,

$$\begin{aligned}
T\llbracket \lambda x.Mx \rrbracket &= \wp(P \cdot \otimes(!\langle D(I) \rangle, I)) \\
&= P \cdot \wp(\otimes(I, I)) \\
&= P \cdot I = P \\
&= T\llbracket M \rrbracket
\end{aligned}$$

□

Proposition 3.3.2 For the $D = \langle D \multimap D \rangle$ translation we have:

1. $\beta : (\lambda x.M)V \rightarrow M[V/x]$ is valid.
2. $\eta : \lambda x.Mx \rightarrow M$ is valid.

◇

PROOF:

1. Straightforward from the definitions:

$$\begin{aligned}
T[(\lambda x.M)V] &= !(\wp(P)) \cdot_z D(\otimes(I, Q)) \\
&\rightarrow \wp(P) \cdot_z \otimes(I, Q) \\
&\rightarrow P \cdot I \cdot Q \\
&\rightarrow P \cdot Q \\
&= T[M[V/x]]
\end{aligned}$$

To show substitution $M[V/x]$ is simulated, we first need to observe that values (V) are of the form $!(-)$ which allows the commutative conversion to take place.

- (a) If $M \equiv x$, then $T[x[V/x]] = I \cdot R = R = T[V]$.
- (b) If $M \equiv \lambda y.A$, then

$$\begin{aligned}
T[M[V/x]] &= !(\wp(P)) \cdot Q \\
&= !(\wp(P) \cdot Q) \\
&\equiv !(\wp(P \cdot Q)) \\
&= T[\lambda x.A[V/x]]
\end{aligned}$$

- (c) If $M \equiv AB$, then assuming $x \in \text{fv}(A)$,

$$\begin{aligned}
T[(AB)[N/x]] &= (P \cdot D(\otimes(Q, I))) \cdot_x R \\
&\equiv (P \cdot_x R) \cdot D(\otimes(Q, I)) \\
&= T[(A[N/x])B]
\end{aligned}$$

- (d) If $M \equiv [x = \langle y, z \rangle]A$ then,

$$\begin{aligned}
T[(x = \langle y, z \rangle)A][N/x] &= C_x^{y,z}(P) \cdot_x R \\
&\rightarrow C(P \cdot R) \\
&= T[[\vec{x} = \langle \vec{y}, \vec{z} \rangle]A][N/x]
\end{aligned}$$

2. To show $\eta, T[\lambda x.Mx] = !(\wp(P \cdot D(\otimes(I, I)))) = P \cdot !(D(I)) = P \cdot I = P = T[M]$.

□

Proposition 3.3.3 For the $D = !D - \circ !D$ translation we have:

1. $\beta : (\lambda x.M)N \rightarrow M[V/x]$ is valid.
2. $\eta : \lambda x.Mx \rightarrow M$ is valid.

◇

PROOF:

1. Straightforward from the definitions:

$$\begin{aligned}
T\llbracket(\lambda x.M)N\rrbracket &= !(\wp(P)) \cdot_z D(\otimes(Q, !(D(I)))) \\
&\rightarrow \wp(P) \cdot \otimes(Q, !(D(I))) \\
&\rightarrow P \cdot Q \cdot !(D(I)) \\
&\rightarrow P \cdot Q \cdot I = P \cdot Q \\
&= T\llbracket M[N/x] \rrbracket
\end{aligned}$$

To show substitution $M[V/x]$ is simulated:

- (a) If $M \equiv x$, then $T\llbracket x[V/x] \rrbracket = !(D(I)) \cdot_x R \rightarrow I \cdot R = R$.
2. To show η , $T\llbracket \lambda x.Mx \rrbracket = !(\wp(P \cdot D(\otimes(!(D(I)), !(D(I)))))) = P \cdot !(D(I)) = P \cdot I = P = T\llbracket M \rrbracket$.

□

Restricted reductions

If we look at our reduction rules for LRA's we see that the context rule permits reductions under *all* contexts — in particular within a “!” context. Computationally however, it is fruitful to restrict contexts and disallow reduction within a box, hence a *lazy* notion of evaluation is implied.

1. A box may be erased, in which case we do not want to waste computation on evaluating its contents.
2. A box may be copied, in which case we would like to share the result.

We now look at this lazy reduction on Linear Logic proof structures, and see what sort of evaluations are induced on λ -terms under the various translations presented.

We begin by stating the results of this section.

- $D = !D \multimap D$ corresponds to an outermost call-by-name reduction to (principal) head normal form.
- $D = !(D \multimap D)$ corresponds to an outermost call-by-value reduction to weak head normal form.

To make these statements more precise, we state the following sequence of results for each translation, for *closed* terms.

Proposition 3.3.4

1. For the $D = !D \multimap D$ translation, if $M \in \Lambda_\sigma$ is in head normal form, then $T\llbracket M \rrbracket$ is in normal form.
2. For the $D = !(D \multimap D)$ translation, if $M \in \Lambda_\sigma$ is in weak head normal form, then $T\llbracket M \rrbracket$ is in normal form.

◇

PROOF:

1. Recall from Definition 2.2.2 that a head normal form is of the following shape:
 $M \equiv \lambda \bar{x}. x \bar{M}$. Now, $T\llbracket M \rrbracket = \wp(\dots \wp(D(I) \cdot \otimes(!(\dots), I)) \dots)$ which is in normal form.
2. Recall from Definition 2.2.3 that a weak head normal form is either $\lambda x.M$.
 Now, $T\llbracket \lambda x.M \rrbracket = !(\wp(P))$ is trivially in normal form.

□

There are many other restrictions that one could impose on the reductions in Linear Logic proof structures, some of which are actually quite useful. We mention just one which is a restriction on the contraction cut:

$$C(P) \cdot_z !(Q)$$

This will copy any redex within Q if we do not allow reduction under $!$. We would prefer this operation to be defined *only if* Q is in full normal form.

3.4 PCF in Linear Logic proof structures

The multiplicative pure calculus tells us how to move *information* around and re-structure it. For example, the linear λ -calculus can be modelled by the B , C , and I combinators which do nothing more than change the order of arguments to a function. In other words the computational power of this system can do nothing more than return the data we gave as input, but maybe structured in different way. The addition of the exponentials then allows the copying and discarding of this *information*. These operations can be characterised by the W and K combinators that we saw in subsection 2.2.5.

In this section we now add this *information* in terms of the constants of PCF. For this inclusion we switch to a restricted system where the λ -terms are required to be *typed*, as given in subsection 2.2.4. All the translations that we have presented work in exactly the same way except that we restrict to a domain of typable terms.

The philosophy that we will follow for coding PCF constants into proof structures is as follows:

- Add the constants, *verbatim*. In particular, a linear constant will be translated into a linear constant in the proof net structure.
- For each translation of the λ -calculus presented, it will be shown how to package the PCF constants so that the translation reflects the given translation.

To fulfill the first part, we extend the Linear Realisability Algebras as follows.

Syntax

For each PCF constant we introduce a new LRA operation. We indicate the free names (fn) and the principal port (pp) for each constant:

Operation	Free Names (fn)	pp
$n_x, \mathbf{tt}_x, \mathbf{ff}_x$	x	x
$\mathbf{succ}_{x,y}, \mathbf{pred}_{x,y}, \mathbf{iszero}_{x,y}$	$\{x, y\}$	x
$\mathbf{cond}_{b,x,y,z}$	$\{b, x, y, z\}$	b
$Y_{x,y}$	$\{x, y\}$	x

Dynamics

The rewrite rules for the LRA constants follow exactly the PCF delta rules and are as expected:

1. $\mathbf{succ}_{x,y} \cdot_x n_x = (n + 1)_y$
2. $\mathbf{pred}_{x,y} \cdot_x 0_x = 0_y$
3. $\mathbf{pred}_{x,y} \cdot_x n_x = (n - 1)_y$
4. $\mathbf{iszero}_{x,y} \cdot_x 0_x = \mathbf{tt}_y$
5. $\mathbf{iszero}_{x,y} \cdot_x n_x = \mathbf{ff}_y$
6. $\mathbf{cond}_{b,x,y,z} \cdot_b \mathbf{tt}_b = W_y(D_x^{x'}(I_{x',z}))$
7. $\mathbf{cond}_{b,x,y,z} \cdot_b \mathbf{ff}_b = W_x(D_y^{y'}(I_{y',z}))$
8. $Y_{x,y} \cdot_x !_x^z(\mathcal{G}_z^{u,v}(F)) = F[y/v] \cdot_u (Y_{x,y} \cdot_x !_x^z(\mathcal{G}_z^{u,v}(F)))$

We can now assign these constants to proofs in a standard way.

Constants at base type

Each constant at base type is introduced as the following linear proofs:

$$\frac{}{n_x \vdash ; x : \mathbf{nat}} \quad \frac{}{\mathbf{tt}_x \vdash ; x : \mathbf{bool}} \quad \frac{}{\mathbf{ff}_x \vdash ; x : \mathbf{bool}}$$

There is an obvious net notation that we will use. For each constant $c : B$ we draw:

$$\begin{array}{c} c \\ | \\ B \end{array}$$

Arithmetic functions

All of the arithmetic functions are linear in their arguments, so we introduce them to reflect this.

$$\frac{}{\mathbf{succ}_{x,y} \vdash x : \mathbf{nat}^\perp ; y : \mathbf{nat}} \quad \frac{}{\mathbf{pred}_{x,y} \vdash x : \mathbf{nat}^\perp ; y : \mathbf{nat}}$$

$$\frac{}{\mathbf{iszero}_{x,y} \vdash x : \mathbf{nat}^\perp ; y : \mathbf{bool}}$$

We will draw these as proof nets in the following manner, where we write f for $\mathbf{succ}$ and $\mathbf{pred}$.

$$\begin{array}{ccc} & f & \\ \diagup & & \diagdown \\ \mathbf{nat}^\perp & & \mathbf{nat} \end{array} \quad \begin{array}{ccc} & \mathbf{iszero} & \\ \diagup & & \diagdown \\ \mathbf{nat}^\perp & & \mathbf{bool} \end{array}$$

Conditional

In linear logic, the type for a conditional is $\mathbf{bool} \otimes (A \wp A) \multimap A$; where $\wp$ is the linear connective *with*.

$$\frac{\vdash \Gamma, A \quad \vdash \Gamma, B}{\vdash \Gamma, A \wp B}$$

Note in particular, with contrast to the rule for $\otimes$, that the context Γ is the same in each premise.

In fact the conditional shows clearly the linear decomposition of cartesian product as $\otimes$ and $\&$ depending on how the values will be used; for a conditional it is linear in the boolean test, and non-linear in the selection of arguments—exactly one branch will be selected.

However, in this thesis we will try to work without the additive fragment of Linear Logic and use a property $!A \otimes !B \multimap A \& B$:

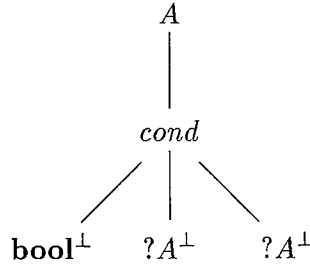
$$\frac{\frac{\frac{\vdash A^\perp, A}{\vdash ?A^\perp, A} \quad \frac{\vdash B^\perp, B}{\vdash ?B^\perp, B}}{\vdash ?A^\perp, ?B^\perp, A} \quad \frac{\vdash ?A^\perp, ?B^\perp, B}{\vdash ?A^\perp, ?B^\perp, B}}{\frac{\vdash ?A^\perp, ?B^\perp, A \& B}{\vdash ?A^\perp \wp ?B^\perp, A \& B}} \quad \frac{}{\vdash (?A^\perp \wp ?B^\perp) \wp (A \& B) = !A \otimes !B \multimap A \& B}$$

So $!A \otimes !B$ implements $A \& B$ by weakening one side, and derelicting the other.

Hence, we take the type of the conditional to be $\mathbf{bool} \multimap (!A \otimes !A) \multimap A$.

$$\overline{\mathbf{cond}_{b,x,y,z} \vdash b : \mathbf{bool}^\perp, x : ?A^\perp, y : ?A^\perp ; z : A}$$

We draw this as a proof net in the following way, changing the orientation so that the result is at the top, and the arguments are along the bottom.



The dynamics of the delta rules for the conditional will decide which argument to weaken, and which to derelict by completing the proof with either:

$$\frac{\frac{I_{u,z} \vdash u : A^\perp ; z : A}{D_x^u(I_{u,z}) \vdash x : ?A^\perp ; z : A}}{W_y(D_x^u(I_{u,z})) \vdash x : ?A^\perp, y : ?A^\perp ; z : A}$$

if the true branch is selected, or

$$\frac{\frac{I_{u,z} \vdash u : A^\perp ; z : A}{D_y^u(I_{u,z}) \vdash y : ?A^\perp ; z : A}}{W_x(D_y^u(I_{u,z})) \vdash x : ?A^\perp, y : ?A^\perp ; z : A}$$

if the false branch is selected.

Recursion

The recursion operator is the only non-linear constant in PCF.

$$\overline{Y_{x,y} \vdash x : ?(!A \otimes A^\perp) ; y : A}$$

It is possible in fact to define $Y_{x,y}$ in the syntax of LRA's, although it is not a typable proof structure of course. We write Y as an abbreviation for the LRA term $\otimes_x^{v,z}(C_{v'}^{y,z'}(I_{v,v'}), I_{z,z'})$. (We encourage the reader to draw this as a “proof net” and compare with the encoding of recursion in graph reduction [Pey87].)

Cut

Here we show a selection of cases of the equations in context.

$$\frac{\text{succ}_{x,y} \vdash x : \text{nat}^\perp ; y : \text{nat} \quad 0_x \vdash x : \text{nat}}{\text{succ}_{x,y} \cdot_x 0_x = 1_y \vdash ; y : \text{nat}}$$

$$\frac{\text{cond}_{b,x,y,z} \vdash b : \text{bool}^\perp, x : ?A^\perp, y : ?A^\perp ; z : A \quad \text{tt}_b \vdash b : \text{bool}}{\text{cond}_{b,x,y,z} \cdot_b \text{tt}_b = W_y(D_x^{x'}(I_{x',z})) \vdash x : ?A^\perp, y : ?A^\perp ; z : A}$$

To fulfill the second part of our programme to encode PCF into Linear Logic proofs we show how we can pack these constants up in LRA's to reflect the translation of the λ -calculus presented. There are two possible ways to proceed:

- First, we could present the constants as packed axiom links and show a separate coding for the delta rules.
- Alternatively, we can “close up” these axiom links following the translations so that application of a constant function is translated as general application.

There is, of course, very little in this. The latter option will be chosen simply because it allows a slightly simpler presentation.

$A \rightarrow B \Rightarrow !A \multimap B$ **Translation**

Constants at base type

Constants at base type are interpreted trivially, i.e. for each constant $c : B$ we have:

$$\overline{c_x \vdash ; x : B}$$

Arithmetic functions

The constant functions **succ** and **pred** get types $!nat \multimap nat$, and **iszero** gets the type $!nat \multimap bool$. We build each of these as follows:

$$\frac{\frac{\text{succ}_{x,y} \vdash x : nat^\perp ; y : nat}{D_z^x(\text{succ}_{x,y}) \vdash z : ?nat^\perp ; y : nat}}{\wp_v^{z,y}(D_z^x(\text{succ}_{x,y})) \vdash ; v : ?nat^\perp \wp nat = !nat \multimap nat}$$

Similarly for **pred** and **iszero**.

Conditional

A conditional gets the type $!bool \multimap !A \multimap !A \multimap A$, which is built as follows:

$$\frac{\frac{\frac{\text{cond}_{b,x,y,z} \vdash b : bool^\perp, x : ?A^\perp, y : ?A^\perp ; z : A}{D_{b'}^b(\text{cond}_{b,x,y,z}) \vdash b' : ?bool^\perp, x : ?A^\perp, y : ?A^\perp ; z : A}}{\wp_u^{y,z}(D_{b'}^b(\text{cond}_{b,x,y,z})) \vdash b' : ?bool^\perp, x : ?A^\perp ; u : ?A^\perp \wp A}}{\wp_v^{x,u}(\wp_u^{y,z}(D_{b'}^b(\text{cond}_{b,x,y,z}))) \vdash b' : ?bool^\perp ; v : ?A^\perp \wp (?A^\perp \wp A)}{\wp_w^{b',v}(\wp_v^{x,u}(\wp_u^{y,z}(D_{b'}^b(\text{cond}_{b,x,y,z})))) \vdash ; w' : ?bool^\perp \wp (?A^\perp \wp (?A^\perp \wp A))}$$

Recursion

The constant Y that we have introduced is already of the right type under this translation: $Y : !(A \multimap A) \multimap A$

$$\frac{Y_{x,y} \vdash x : !(A \otimes A^\perp) ; y : A}{\wp_z^{x,y}(Y_{x,y}) \vdash ; z : !(A \multimap A) \multimap A}$$

$A \rightarrow B \Rightarrow !(A \multimap B)$ Translation

3.4.1 Constants at base type

Constants at base type are interpreted as promoted values, i.e. for each constant $c : B$ we have:

$$\frac{c_x \vdash ; x : B}{!^x_y(c_x) \vdash ; y : !B}$$

Arithmetic functions

The constant functions **succ** and **pred** get types $!(\text{nat} \multimap \text{nat})$, and **iszero** gets the type $!(\text{nat} \multimap \text{bool})$. We build each of these in the following way:

$$\frac{\frac{\frac{\text{succ}_{x,y} \vdash x : \text{nat}^\perp ; y : \text{nat}}{D_z^x(\text{succ}_{x,y}) \vdash z : ?\text{nat}^\perp ; y : \text{nat}}}{!^y_u(D_z^x(\text{succ}_{x,y})) \vdash z : ?\text{nat}^\perp ; u : !\text{nat}}}{\mathfrak{I}_v^{z,u}(!^y_u(D_z^x(\text{succ}_{x,y}))) \vdash ; v : ?\text{nat}^\perp \mathfrak{I} !\text{nat}}}{!^v_w(\mathfrak{I}_v^{z,u}(!^y_u(D_z^x(\text{succ}_{x,y})))) \vdash ; w : !(? \text{nat}^\perp \mathfrak{I} !\text{nat}) = !(\text{nat} \multimap \text{nat})}$$

Similarly for **pred** and **iszero**.

Conditional

The conditional gets the type $!(\text{bool} \multimap !(A \multimap !(A \multimap A)))$, which we build as follows:

$$\frac{\frac{\frac{\text{cond}_{b,x,y,z} \vdash b : \text{bool}^\perp, x : ?A^\perp, y : ?A^\perp ; z : A}{D_{b'}^b(\text{cond}_{b,x,y,z}) \vdash b' : ?\text{bool}^\perp, x : ?A^\perp, y : ?A^\perp ; z : A}{!^z_v(D_{b'}^b(\text{cond}_{b,x,y,z})) \vdash b' : ?\text{bool}^\perp, x : ?A^\perp, y : ?A^\perp ; u : !A}{\mathfrak{I}_v^{y,u}(!^z_v(D_{b'}^b(\text{cond}_{b,x,y,z}))) \vdash b' : ?\text{bool}^\perp, x : ?A^\perp ; v : ?A^\perp \mathfrak{I} !A}{!^v_{v'}(\mathfrak{I}_v^{y,u}(!^z_v(D_{b'}^b(\text{cond}_{b,x,y,z})))) \vdash b' : ?\text{bool}^\perp, x : ?A^\perp ; v' : !(?A^\perp \mathfrak{I} !A)}{\mathfrak{I}_w^{x,v'}(!^v_{v'}(\mathfrak{I}_v^{y,u}(!^z_v(D_{b'}^b(\text{cond}_{b,x,y,z})))))) \vdash b' : ?\text{bool}^\perp ; w : ?A^\perp \mathfrak{I} !(?A^\perp \mathfrak{I} !A)}{!^w_w(\mathfrak{I}_w^{x,v'}(!^v_{v'}(\mathfrak{I}_v^{y,u}(!^z_v(D_{b'}^b(\text{cond}_{b,x,y,z})))))) \vdash b' : ?\text{bool}^\perp ; w' : !(?A^\perp \mathfrak{I} !(?A^\perp \mathfrak{I} !A))}{\mathfrak{I}_r^{b',w'}(!^w_w(\mathfrak{I}_w^{x,v'}(!^v_{v'}(\mathfrak{I}_v^{y,u}(!^z_v(D_{b'}^b(\text{cond}_{b',x,y,z})))))) \vdash ; r : ?\text{bool}^\perp \mathfrak{I} !(?A^\perp \mathfrak{I} !(?A^\perp \mathfrak{I} !A))}{!^r_s(\mathfrak{I}_r^{b',w'}(!^w_w(\mathfrak{I}_w^{x,v'}(!^v_{v'}(\mathfrak{I}_v^{y,u}(!^z_v(D_{b'}^b(\text{cond}_{b',x,y,z})))))) \vdash ; s : !(\text{bool}^\perp \mathfrak{I} !(?A^\perp \mathfrak{I} !(?A^\perp \mathfrak{I} !A)))}$$

PCF Constant	Translation	pp
c	c_x	x
$f_{x,y}$	$D_z^x(f_{x,y})$	z
$\mathbf{cond}_{b,x,y,z}$	$D_{b'}^b(\mathbf{cond}_{b,x,y,z})$	b'
$Y_{x,y}$	$Y_{x,y}$	x

Table 3.4: $!A \multimap B$ translation

PCF Constant	Translation	pp
c_x	$!_y^x(c_x)$	y
$f_{x,y}$	$!_v^y(D_u^x(f_{x,y}))$	v
$\mathbf{cond}_{b,x,y,z}$	$!_u^z(D_{b'}^b(\mathbf{cond}_{b,x,y,z}))$	u
$Y_{x,y}$	$Y_{x,y}$	x

Table 3.5: $!(A \multimap B)$ translation

Recursion

$$\frac{\frac{Y_{x,y} \vdash x : ?(!A \otimes A^\perp) ; y : A}{\wp_w^{x,y}(Y_{x,y}) \vdash ; w : !(!A \multimap A) \multimap A}}{!_z^w(\wp_w^{x,y}(Y_{x,y})) \vdash ; z : !(!A \multimap A) \multimap A)}$$

In Tables 3.4 and 3.5 we give a summary of the translations given for the PCF constants. In particular, we indicate the principal port for each construct.

For the second translation, the one that we will use most often, we show some subtleties in the cut elimination process. First we discuss the arithmetic functions, then examine the conditional.

The delta rule $\mathbf{succ} \ n = n + 1$ is given by the following sequence of reductions:

$$\begin{aligned} \mathbf{succ} \ n &= !_v^y(D_{x'}^z(\mathbf{succ}_{z,y})) \cdot_{x'} !_x^x n_x \\ &= !_v^y((D_{x'}^z(\mathbf{succ}_{z,y})) \cdot_{x'} !_x^x(n_x)) \\ &= !_v^y((\mathbf{succ}_{x,y}) \cdot_x n_x) \\ &= !_v^y(n+1)_y \end{aligned}$$

Hence we needed to use the commutative cut to complete the reduction sequence. This will be of interest when we look at the interaction net implementation in Chapter 5.

The conditional throws up additional problems. The delta rule $\mathbf{cond} \ \mathbf{tt} \ M \ N = M$ is given by the following sequence of reductions:

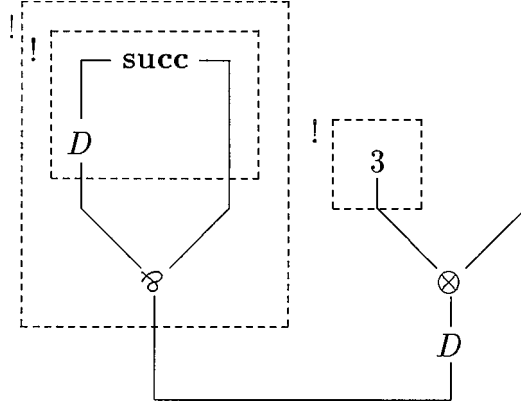
$$\begin{aligned}
 \mathbf{cond} \ \mathbf{tt} \ M \ N &= !_v^z(D_{b'}^b(\mathbf{cond}_{b,x,y,z})) \cdot_{b'} !_{b'}^b(\mathbf{tt}_b) \\
 &= !_v^z(D_{b'}^b(\mathbf{cond}_{b,x,y,z}) \cdot_{b'} !_{b'}^b(\mathbf{tt}_b)) \\
 &= !_v^z(\mathbf{cond}_{b,x,y,z} \cdot_b \mathbf{tt}_b) \\
 &= !_v^z(W_y(D_x^{x'}(I_{x',z})))
 \end{aligned}$$

Now, by the η -rule for LRA's (Section 2.4.1) this reduces to $W_y(I_{v,x})$.

Hence, in addition to using the commutative conversion to complete the reduction we also required the use of the η -rule. Again, this will be of particular interest when we consider the implementation.

Example 3.4.1 Here we give an example net for the $!(A \multimap B)$ translation that we will use several times in this thesis.

The PCF term $(\lambda x.\mathbf{succ} \ x)3$ can be drawn as follows:



◇

Remark 3.4.2 We could have chosen a very different philosophy for the encoding of PCF into Linear Logic proof structures, by translating the constants into Linear $\lambda\sigma$ -terms. For example one such translation is given by:

tt	=	$\lambda xy.[y = _]x$
ff	=	$\lambda xy.[x = _]y$
$\bar{0}$	=	$\lambda fx.[f = _]x$
$\bar{1}$	=	$\lambda fx.fx$
$\bar{n}$	=	$\lambda fx.[f = \langle f_1, f \rangle] \cdots [f = \langle f_{n-1}, f \rangle] f_1(\cdots f_{n-1}(fx))$
succ	=	$\lambda abc.[b = \langle b_1, b_2 \rangle] b_1(ab_2c)$
pred	=	$\lambda z.YHzS^+I\mathbf{ff}$
cond	=	$\lambda xyz.xyz$
iszero	=	$\lambda n.nS^+I\mathbf{tt}$
Y	=	Θ

Where I is the identity function, H and S^+ are defined as:

$$\begin{aligned} H &= \lambda hx.\mathbf{iszero} \ x \ \bar{0} \ (\mathbf{succ}(h(x\mathbf{ff}))) \\ S^+ &= \lambda xy.y\mathbf{ff}x \end{aligned}$$

and Θ , Turing's fixed point combinator, is defined as:

$$(\lambda xy.[x = \langle x_1, x_2 \rangle][y = \langle y_1, y_2 \rangle]y_1(x_1x_2y_2))(\lambda xy.[x = \langle x_1, x_2 \rangle][y = \langle y_1, y_2 \rangle]y_1(x_1x_2y_2))$$

Hence it is possible to program in the pure $\lambda\sigma$ -calculus and implement various data types by coding tricks. We prefer, however, to add several constants to the calculus for both practical and efficiency purposes. Turner [Tur91] gives a good discussion on the relative merits of adding constants to a pure calculus. $\diamond$

3.5 Discussion

We have presented three different translations of the pure λ -calculus into Linear Logic proof structures. With respect to our programme of research, we would like to choose one which gives “more sharing”. Of course, this is going to depend on which implementation we are considering. In [GAL92] we find the statement:

...the standard translation of the λ -calculus in Linear Logic, based in $D = !D \multimap D$, does not have this problem because it enforces less sharing.

There is now the question to which translation is “better” than another, and indeed to ask how many more translations exist, and are any of these of interest?

We propose the following. Since the only translation that allows us to use the η -rule for one of the compilation rules is the $D =!(D \multimap D)$ translation, this is a compilation strategy which yields a net with the fewest nodes without the need for additional passes over the structure of the proof to perform optimisations, and hence has less cut elimination steps.

For this reason alone we choose to use this translation throughout the rest of this thesis, except where we add a comment about one of the other translations. There are also other issues pointed out to me by Gonthier [personal communication] concerning Garbage collection issues that favours this translation. Also, this translation, as we have seen, corresponds to a very standard notion of reduction used in functional language implementation, namely reduction to weak head normal form.

With respect to different translations, there are many, and few are of significant interest. We have tried to identify three which say something about known reductions in the λ -calculus. Note that the translations that we use are *safe* in the sense that they make “everything non-linear”. One may ask for a more refined translation in which the only non-linear parts have boxes around them. For example $\lambda x.\lambda y.x$ is only non-linear in y but all the translations would treat x and y the same. This issue has been studied in [Rov92], for example, and a more systematic treatment for Linear Logic has been studied in [Sch94] where a notion of *decoration* is considered. We remark finally that detecting linearity is trivially undecidable:

$$\lambda P.\lambda x. \text{ if halts}(P) \text{ then } x \text{ else } x + x.$$

One can only ever make some notion of approximation to this problem.

Chapter 4

Paths, Labels and Types

This chapter takes a detailed look at the notion of a path in a λ -term. We show some properties of paths and show how they relate to labels and types.

4.1 Introduction

In this chapter we set out the theoretical foundation of Linear Logic proof structures by providing a mathematical model in which we can encode these proofs (programs). We are faced with a wide spectrum of semantic paradigms on which we could base our reasoning. Until recently these roughly fell into three distinct classifications: *operational*, *axiomatic* and *denotational*.

It is the belief of the author that the choice of which paradigm to select depends on what the application of the semantics is:

- Operational semantics is biased towards the language implementer in that each atomic computation step is specified and can be directly implemented. Hence the paradigm is *dynamic* in the sense that it specifies how the computation will proceed.
- Axiomatic semantics is biased towards the user of the language in that it provides a way of proving abstract properties about the program that is free from any implementation.
- Denotational semantics is biased towards the language designer in that it provides a means to specify what should be the effect of a computation rather

than how it is obtained. In particular the paradigm is *static* in that we are only dealing with notions of answers of a computation.

At this point we remind the reader of our objective of using a semantics *directly* as an implementation technique. We emphasise the word “directly” because of the desire to avoid additional encoding for an implementation and to eliminate the necessity of correctness proofs. From the above description of the semantic paradigms this clearly indicates that we should be investigating an operational account of the cut-elimination process of Linear Logic. We gave such an account in Chapter 2 where we specified the rewriting rules for LRA’s which are representations of Linear Logic proofs. However there is an unfortunate aspect of calculi of this form in that the names (cf. variables in the λ -calculus) play a crucial rôle in the rewriting process. There have been attempts to try to mathematically analyse this renaming process, for example this is exactly the purpose of the de Bruijn λ -calculus [deB72] which has been successfully used as an implementation technique in the Categorical Abstract Machine [Hue90].

This discussion leads to what the author believes to be a rather strange phenomenon with respect to the (author’s) opinion that operational semantics is biased towards the language implementor. It appears that the first thing that an implementor is required to do is *formalise* the operational account — in other words try to add denotational features to the operational setting.

However, trying to formalise a complicated operation such as substitution leads to a complicated mathematical account. This can be observed simply by looking at the number of axioms that are required in the Linear $\lambda\sigma$ -calculus for example.

In the light of these issues, we believe that what is really needed is a new perspective on these semantics to find a *natural* middle ground between the operational and denotational world rather than incorporating seemingly random mathematical tools into the operational account.

Girard, in a series of papers [Gir89b, Gir89a, Gir88, Gir94], embarked on a radically new approach to semantics under the name *the Geometry of Interaction*. His motivations were to provide mathematical tools for the study of the cut-elimination process in Linear Logic proof structures; but the fundamental ideas are applicable to computation in general. The prominent features of his programme can be summarised as follows.

- The semantics is *syntax free* in that it avoids the *bureaucratic* issues that come with the notion of substitution (cf. the λ -calculus).
- The semantics is chiefly a study of the dynamics of the cut-elimination process

(hence β -reduction) using denotational tools. The model is termed *dynamic* since it captures the essential computational content.

- The semantics is operational in the sense that there are a set of prescribed rules specifying how computation should proceed at each step.

It is hoped that this new understanding of the dynamics of computation could lead to radically new implementation techniques for programming languages, and with respect to the above discussion it fits very strongly with our view. We hope that this thesis provides a contribution towards this goal.

Hence the semantics of Linear Logic proof structures that we use will be the Geometry of Interaction. This will be used in two very different ways in this thesis:

- First, in Chapter 5, we will use it as an aid to showing the correctness of a graph rewriting algorithm for the λ -calculus.
- On the other hand, in Chapter 6, we take a completely different perspective and give an implementation of this semantics directly.

The purpose of this chapter then is first to give a semantics of Linear Logic in terms of Girard's *Geometry of Interaction*. Our presentation will be based on the notion of a *path* in a proof structure which follows the work of Danos [Dan90] and Regnier [Reg92]. We begin with our presentation of a path in the λ -calculus, and review the various notions that can be found in the literature. We then extend this notion to give paths for PCF. Finally, we look at the theory and investigate some useful properties that we can take advantage of later in this thesis. We conclude the chapter by hinting at some connections between paths, labels and types.

4.2 Proofs and Paths

The philosophy behind this semantic paradigm is a notion of a set of *paths* in a program. An algebraic device is used which gives a notion for how sets of paths can be joined together, and computation is given by calculating the transitive closure of these paths. All the work that is required in this interpretation is to select which paths are the “good ones”, and eliminate the rest.

Proof nets are graphs $G = (V, E)$ in the usual sense [Big89]. The *vertices* V are the logical symbols (for example $\otimes$ and $\wp$ in the multiplicative fragment) and the *edges* E are the connecting links that join these symbols to their premises and conclusions.

There is a natural notion of a *walk* (*op. cit*) in a graph which is nothing more than a travel around the structure. A walk in a graph is specified uniquely by a sequence of vertices. There is also a notion of a path in a graph that is defined to be a restricted walk, in particular it traverses every vertex exactly once.

Here we take a slightly different perspective on these standard notions.

- First we label the edges of a proof net rather than the vertices. The labels that we use are elements of an algebra Λ^* ; shortly to be defined.
- Consequently the walks in a proof net are specified by a sequence of edges rather than vertices.
- The notion of a path that we shall use will be restricted by the algebra Λ^* .

We begin with the presentation for the λ -calculus, then go on to extend the notions to PCF.

Definition 4.2.1 (Labels)

The atomic labels l that we use for the proof net structure are taken from the set: $\{0, 1, p, q, r, s, t, d\}$. $\diamond$

Definition 4.2.2 A labelled proof net is a *directed* graph $G = (V, w)$ where

- V is the set of vertices of the proof net structure drawn from $\{\otimes, \wp, D, C, W\}$.
- w is a set of atomically labelled directed edges called its *weight*. If e is an edge from vertex x to y with label l then we write $w(e) = l$; if e is an edge from y to x then we write $w(e) = l^*$ — the reverse of l .

$\diamond$

Definition 4.2.3 A walk ϕ in a proof net is a sequence of weights that we write from right to left (*cf. left-hand* notation for function composition) such that it satisfies the following conditions:

Non-bouncing A walk is said to be *bouncing* if it returns on itself.

i.e. there are no subsequences of the form $\phi^* \phi$.

Non-twisting A walk in a proof net is said to be *twisting* if the travel arrives from one premise of a binary link and continues into the other premise.

If ϕ is a walk from vertex x to y , then we extend the reverse notation in a natural way and write ϕ^* for a walk from y to x . which satisfies $(\phi_1 \phi_2)^* = \phi_2^* \phi_1^*$.

The set of possible walks from a given edge in the graph is a prefixed closed sum of walks. $\diamond$

Walks can start anywhere, and we can grow them in any direction. For a proof net structure the start and the end of a walk will be the conclusions of the proof and, as we shall shortly see, weakening nodes.

We now present the labelled proof net structures, drawing arrows on the labelled edges to indicate the direction of the edge. We also give some examples of walks in the structure.

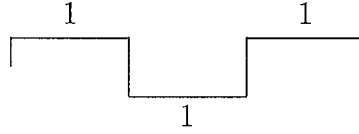
Identity group

The Axiom and Cut links are connecting edges that we draw in an *undirected* way. We label these identities with 1 which are respectively drawn as:



Hence we have $1^* = 1$.

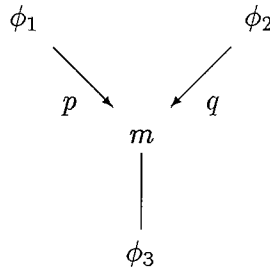
For example, a cut of two axiom links yields the following labelled net:



which allows the walk in the net: 111.

Multiplicative group

We collapse the multiplicative connectives in this semantic paradigm, hence it is assumed that we are working with *correct* proof nets as discussed in Chapter 2. With the intuition that p means “left” and q means “right”, the following diagram shows how we label the multiplicative nodes $m \in \{\otimes, \wp\}$ ¹.

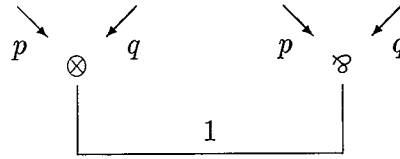


¹We could refine the presentation and replace p and q by $p_{\wp}, p_{\otimes}$ and $q_{\wp}, q_{\otimes}$ respectively, however this is not necessary.

We identify the possible walks in this structure.

- The walk ϕ_3 arriving at the conclusion can continue in two possible directions giving the walks $p^*\phi_3$ and $q^*\phi_3$ — extending along the p and q edges in the *negative* direction respectively. We write this as a disjoint sum of walks $(p^* + q^*)\phi_3$.
- If ϕ_1 is a walk arriving from the left premise, then the unique extension is a walk $p\phi_1$ and continuing out at the conclusion. Note that the restriction of a *non-twisting* walk eliminates the possibility to extend along q^* .
- The case for a walk arriving from the right premise is similar.

The multiplicative cut of a tensor against a par:



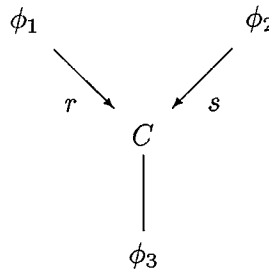
allows several walks in the net. For example starting from the left premise of the $\otimes$ we have two possible walks:

1. p^*1p
2. q^*1p

Exponential group

Contraction

Contraction is entirely similar to a multiplicative except that we use r and s to label the left and right premises respectively.



Weakening

Weakening marks the end of a walk where no further extensions are possible. In our implementation in Chapter 6 we will see that these walks are *never* accessed, so we pay little attention to them in the theory. However, for completeness, we give the labelling:

$$\begin{array}{c} W \\ | \\ 0 \\ | \\ \phi \end{array}$$

If ϕ is a walk arriving at a weakening node, then the walk ends with 0ϕ .

Dereliction

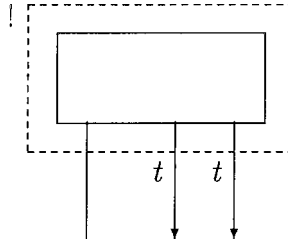
Dereliction is labelled by the d operator of the algebra:

$$\begin{array}{c} \phi_1 \\ \downarrow d \\ D \\ | \\ \phi_2 \end{array}$$

- If ϕ_2 is a walk arriving from the conclusion, then the only possible extension is a walk $d^*\phi_2$.
- If ϕ_1 is a walk arriving from the unique premise, then the only possible extension is a walk $d\phi_1$.

Promotion

The labelling of the promotion rule is slightly more complicated because of the box structure. We label a box as follows:



There are two features:

1. Walks within a box are *lifted* by an $!$ operator. Each label l is lifted $!(l)$, and again there is an obvious extension of this to walks satisfying $!(\phi_1)!(\phi_2) = !(\phi_1\phi_2)$.
2. The side doors of the box are labelled with t . Hence if $!(\phi)$ is a walk within a box that exits along a side door edge we write $t!(\phi)$.

In the absence of a walk arriving at a weakening we can identify three types of walk for a box structure:

Type I Entering through the principal port, and returning back through this port. An example of such a walk would be in the term $!_u^z(\wp_z^{x,y}(I_{x,y}))$ where both walks are of this form, i.e. passing through $uzxyzu$ and $zyxzu$.

Type II Entering through the principal port, and exiting from one of the auxiliary doors. For example $!_u^u(\wp_u^{z,x}(W_z(I_{x,y})))$ where there is a walk entering through v and exiting through y .

Type III Entering through an auxiliary door, and returning through an auxiliary door (perhaps even the same one). For example in the net $!_w^{w'}(\wp_w^{z,x'}(W_z(\otimes_v^{x,y}(I_{x,x'}, I_{y,y'}))))$ there is a walk passing along $vy y'$.

These are also called respectively *palpal*, *palpax* and *paxpax* in the literature [DR94]. If we were also considering weakening, we remark that walks will simply enter either port, and not return.

We now need to define the notion of a path which is a restricted walk in the proof net. For this we need to take a little detour in to some algebraic material on the labels. The following definition of the algebra that we use is essentially taken from [Dan90].

Definition 4.2.4 (Λ^*)

Λ^* is a single sorted Σ algebra. We write x and y for the variables, and all other symbols $0, 1, p, q, r, s, t, d : \Sigma$ are constants of the theory. There is a associative multiplication operator $\cdot : \Sigma \times \Sigma \rightarrow \Sigma$, (that we will omit), which has unit 1 and absorbing element 0. The theory is equipped with an involution $*$: $\Sigma \rightarrow \Sigma$ and an exponential operator $! : \Sigma \rightarrow \Sigma$. The following equations define the properties that we require.

$$\begin{array}{ll}
0^* = !(0) = 0 & 1^* = !(1) = 1 \\
0x = x0 = 0 & 1x = x1 = x \\
!(x)^* = !(x^*) & (xy)^* = y^*x^* \\
(x^*)^* = x & !(x)!(y) = !(xy)
\end{array}$$

Annihilation

There are six constants to consider which we have already seen as labels of proof nets: p and q are the *multiplicative* coefficients, r and s are the *contraction* coefficients, d is the *dereliction* coefficient, and finally, t is the *side door* coefficient. The following are the *annihilation* equations; showing how the composition of constants can be simplified.

$$\begin{array}{ll}
p^*p = q^*q = 1 & q^*p = p^*q = 0 \\
r^*r = s^*s = 1 & s^*r = r^*s = 0 \\
d^*d = 1 & t^*t = 1
\end{array}$$

Communication

The exponential coefficients interact with the exponential morphism in the following manner. These are also known as the *leap frog* equations since they all involve the constants jumping over the exponential morphism.

$$\begin{array}{ll}
!(x)r = r!(x) & !(x)s = s!(x) \\
!(x)t = t!(x) & !(x)d = dx
\end{array}$$

Note that the equations for r and s simply jump over the exponential whereas the t operator adds an extra $!$ and the d operator removes a $!$.

We call terms built out of this theory *monomials*. To write *polynomials* we need the notion of a sum of monomials.

Sum

Finally, we show the equations for $+$ which is a formal sum used to allow us to write *disjoint* monomials.

$$\begin{array}{llll}
x + y & = & y + x & x + 0 & = & x \\
(x + y)z & = & (xz + yz) & z(x + y) & = & (zx + zy) \\
(x + y)^* & = & x^* + y^* & !(x + y) & = & !x + !y
\end{array}$$

◇

There is a natural orientation of *monomial* equations, leading to a simple term rewriting system. Moreover, there is a trivial Church-Rosser theorem by the fact that there are no critical pairs. We will look deeper into this theory later in this chapter.

To give an intuition about the monomials of this algebra, we present a simple model, the so-called *small model* which is essentially taken from [Dan90].

Definition 4.2.5 (Small model)

Let $\ulcorner \cdot, \cdot \urcorner$ be a bijection $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, for example $\ulcorner m, n \urcorner = 2^m(2n + 1) - 1$.

We define the carrier of the model to be the partial injective functions on $\mathbb{N}$; so $\llbracket \Sigma \rrbracket = [\mathbb{N} \rightarrow \mathbb{N}]$

The constants of Λ^* are interpreted as $\llbracket 0 \rrbracket, \llbracket 1 \rrbracket, \llbracket p \rrbracket, \llbracket q \rrbracket, \llbracket r \rrbracket, \llbracket s \rrbracket, \llbracket t \rrbracket, \llbracket d \rrbracket : [\mathbb{N} \rightarrow \mathbb{N}]$ as defined below.

The operations are interpreted as:

- $\llbracket \cdot \rrbracket$ is defined to be the composition of partial injective functions $\llbracket \cdot \rrbracket : [\mathbb{N} \rightarrow \mathbb{N}] \times [\mathbb{N} \rightarrow \mathbb{N}] \rightarrow [\mathbb{N} \rightarrow \mathbb{N}]$.
- $\llbracket f^* \rrbracket$ is defined to be the inverse partial injective function $\llbracket f \rrbracket^{-1}$.
- $\llbracket ! \rrbracket$ is defined as: $\llbracket !(f) \rrbracket \ulcorner a, b \urcorner = \ulcorner a, \llbracket f \rrbracket(b) \urcorner$.

We now show the interpretation of the constants, and we omit the semantic brackets to aid readability.

Constants

1 is the identity and 0 is the nowhere defined function on $\mathbb{N}$.

Multiplicatives

The operations p and q are defined on $\mathbb{N}$ as follows:

$$p(n) = 2n \quad q(n) = 2n + 1$$

The duals p^* and q^* are defined to be the natural (partial) inverses.

Exponentials

The exponential coefficients are defined on $\mathbb{N}$ using the bijection to code the “tree structure”:

$$\begin{aligned} r(\ulcorner a, b \urcorner) &= \ulcorner 2a, b \urcorner & s(\ulcorner a, b \urcorner) &= \ulcorner 2a + 1, b \urcorner \\ t(\ulcorner a, \ulcorner b, c \urcorner \urcorner) &= \ulcorner \ulcorner a, b \urcorner, c \urcorner & d(a) &= \ulcorner 0, a \urcorner = 2a \end{aligned}$$

Again, the duals to these operations are defined to be the partial inverses. ◇

Proposition 4.2.6 The above interpretation is a sound model of Λ^* . ◇

PROOF: We need to show that the equations given in Definition 4.2.4 are sound in this model. The multiplicative fragment is very simple to verify, we show just two cases, and again we omit semantic brackets.

$$\begin{aligned} p^*p(n) &= p^*(2n) & p^*q(n) &= p^*(2n+1) \\ &= n & &= \text{undefined} \end{aligned}$$

We also show two exponential cases:

$$\begin{aligned} !(f)!(g)(\ulcorner a, b \urcorner) &= !(f)(\ulcorner a, g(b) \urcorner) & !(f)t(\ulcorner a, \ulcorner b, c \urcorner \urcorner) &= !(f)(\ulcorner \ulcorner a, b \urcorner, c \urcorner) \\ &= \ulcorner a, f(g(b)) \urcorner & &= \ulcorner \ulcorner a, b \urcorner, f(c) \urcorner \\ &= !(fg)(\ulcorner a, b \urcorner) & &= t(\ulcorner \ulcorner a, \ulcorner b, f(c) \urcorner \urcorner) \\ & & &= t!(f)(\ulcorner a, \ulcorner b, c \urcorner \urcorner) \end{aligned}$$

□

We are now in a position to define the notion of path in a proof net structure.

Definition 4.2.7 (Path)

A path ϕ in a proof net is a walk such that $\Lambda^* \vdash \phi \neq 0$. ◇

There are certain kinds of path that we can identify that will be of particular use.

Definition 4.2.8 Let ϕ and ψ be paths. If $\phi\psi \neq 0$ then we call ϕ an *extension* to ψ . If $\psi\phi \neq 0$ then we call ϕ a *co-extension* to ψ .

We say a path ϕ is *maximal* in the case that there are no further extensions/co-extensions to this path. ◇

Remark 4.2.9 Hence a maximal path is one that starts and finishes at a conclusion. For a closed term, all maximal paths start and finish at the root (the only conclusion). ◇

Definition 4.2.10 (Execution path)

Let G be a proof net with conclusions $\Gamma = A_1, \dots, A_n$. The set of execution paths $\mathcal{X}$ in G are all the maximal paths that have domains and codomains in $A_i, 1 \leq i \leq n$. The meaning of a proof net is then given by the disjoint union of these paths.

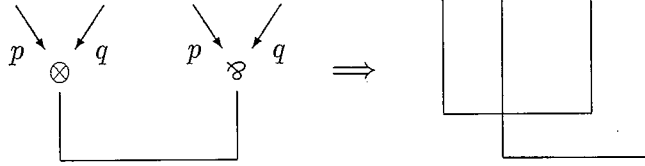
$$\mathcal{E}\mathcal{X}(G) = \bigoplus_{\phi \in \mathcal{X}} \phi$$

Note that execution paths are *maximal* in the sense that they cannot be extended; this is a direct consequence of the definition of a path. ◇

Here we will not justify *why* this theory is the right notion, but we will explain *how* it works. The idea is that a path in a proof net is a “good one” if it survives the action of a cut. The notions that we have presented here captures this precisely. We look at some examples:

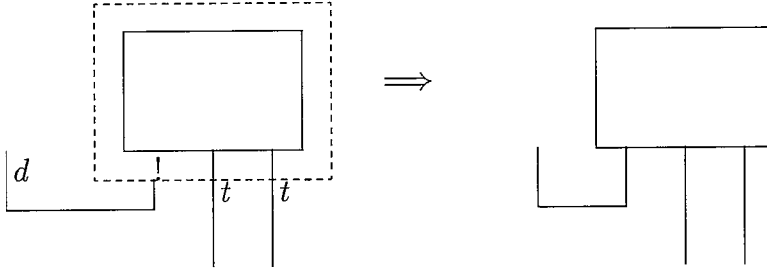
Having set up the notion of a path, we now justify their purpose. For all the cut-elimination rules for Linear Logic we can assign a path before and after the cut. A path semantics for Linear Logic is a technical tool that we can use to show “invariance”; more precisely, the path semantics is used to show that all paths survive the action of a cut. We show a few critical cases.

The multiplicative cut of a tensor against a par:



The only valid path constructions are given by p^*p and q^*q , which both reduce to 1 in the algebra. Hence, these correspond exactly to the resulting paths after cut elimination. The equations $p^*p = 1$ and $q^*q = 1$ together can thus be seen as coding linear β -reduction.

The cut of a dereliction against a box shows an interesting “mismatch” between the path semantics and the cut elimination process.



We have 3 paths to consider:

Type I Before the cut, we have the path $d^*(\phi)d$ which has weight ϕ ; the same weight as the path after the reduction.

Type II Before the cut, we have the path $t!(\phi)d$ which has weight $td\phi$. After the cut we have a path of weight ϕ . Since we do *not* have the equation $td = 1$, there is a mismatch between the path invariance and the cut elimination. This problem has been studied in the context of optimal reduction and corresponds to an accumulation of the *bracket* and *croissant* nodes that leads to one of the main inefficiencies of optimal reducers.

Type III Before the cut, we have the path $t!(\phi)t^*$. After the cut we have the path ϕ . Again we have a mismatch for invariance.

A similar phenomenon arises in the cut of contraction against a box.

We must now justify the use of these paths in the light of above. The solution to this problem can be found in [Dan90] where it is shown that the right property to capture is not equality of terms under reduction, but the *survival* of paths under reduction. This leads to a notion of *persistent path* which we will characterise as follows:

Let ϕ be a path crossing a cut link between vertices v and v' , and ϕ' the path between these vertices after the cut. If $\vdash \phi \neq 0$ then $\vdash \phi' \neq 0$.

This says that paths survive the action of a cut which formalises the intuition that the Geometry of Interaction interpretation gives an understanding of proof nets as electrical circuits which stay connected during reduction.

PCF Constants

We now extend the notion of a path to include the constants of PCF. With this addition the notion of a *question* and *answer* will play a crucial rôle in the semantics. The idea is that a travel along a path is now either a question: looking for a data value, or an answer: returning from a data value.

We follow the same presentation as before, first labelling the PCF extensions of the proof net structure that we gave in Chapter 3 and give examples of walks in the structure. We then extend the algebra to include these constants so that we can define the corresponding notion of a path in a PCF proof net.

We show the interpretation free from any particular choice of translation of these constants into Linear Logic proof structures. Hence we are following the presentation that we gave at the start of Section 3.4. Recall that the different translations only differ by how we *decorate* the constants with exponential information. Hence using the path interpretation of the exponentials we get an interpretation for any translation that works.

First we extend the labels which are just the PCF constants $l = \{\bar{n}, \mathbf{tt}, \mathbf{ff}, \mathbf{succ}, \mathbf{pred}, \mathbf{iszero}\}$. There will not be a label for the recursion as we explain below, and we define the conditional in terms of $\mathbf{tt}$ and $\mathbf{ff}$. As before, these labels are *directed*. If we traverse an edge in the opposite direction we write l^* .

There is one key difference that we point out before giving the presentation. The notion of a *walk* will still be restricted to be *non-bouncing* and *non-twisting*, but the constants of the network will act as *turning points* as we shall make clear below.

Data values

The PCF constants represent terminal nodes of the graph. We label the actual end point of the edge in this case as follows:

$$\begin{array}{c} \llbracket c \rrbracket \\ \downarrow \\ \phi \end{array}$$

The interpretation of each constant c is given by the corresponding label, i.e. $\llbracket c \rrbracket$ is defined as:

$$\begin{array}{ll} \llbracket \mathbf{tt} \rrbracket &= \mathbf{tt} \\ \llbracket \mathbf{ff} \rrbracket &= \mathbf{ff} \\ \llbracket \bar{n} \rrbracket &= \bar{n} \end{array}$$

If ϕ is a walk arriving at a constant, then the only possible way of extending the walk is to turn back on itself along ϕ^* . Hence we write $c\phi$ for the extension to the walk.

We will use the intuition that the walk ϕ is a *question* and c is an *answer* to that question.

Arithmetic functions

For $f \in \{\mathbf{succ}, \mathbf{pred}, \mathbf{iszero}\}$ we have the following labelling:

$$\begin{array}{ccc} & \llbracket f \rrbracket & \\ \text{---} & & \text{---} \\ | & & | \\ \phi_2 & & \phi_1 \end{array}$$

The functions f are interpreted by the corresponding labels where $\llbracket f \rrbracket$ is defined as:

$$\begin{array}{ll} \llbracket \mathbf{succ} \rrbracket &= \mathbf{succ} \\ \llbracket \mathbf{pred} \rrbracket &= \mathbf{pred} \\ \llbracket \mathbf{iszero} \rrbracket &= \mathbf{iszero} \end{array}$$

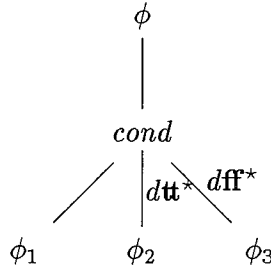
- If ϕ_1 is a walk arriving at the function f (so it is a question asking for the result of f applied to its argument) then we continue the walk in search of the

value of the argument along ϕ_2 with the walk $f\phi_1$ (so we are asking a new question).

- If ϕ_2 is a walk arriving from the argument of the function, then we are answering to the question asked by the function for the value of the argument. Hence, we extend the walk by “applying” the function: $f^*\phi_2$ and continue out of the function back along ϕ_1 to answer the previous question.

Conditional

For the conditional we require a notion of *choice* in the algebra which will give the synchronisation required. In other words, we require the boolean value to be available before we decide which branch of the conditional we need to consider. Hence a walk in the graph should investigate the boolean argument before deciding which branch to follow. We label the graph as follows:



There are three kinds of walk to consider:

- If ϕ is a walk arriving from the top, then we are asking for the result of the conditional. Hence we extend the walk along the edge of the boolean argument; asking a question to the value of the argument.
- If ϕ_1 is a walk arriving from the boolean argument (so the answer to the boolean question), then we can extend the walk along either the true or false branch depending on the value of the test. Hence there are two possible extensions to the walk $dff^*\phi_1$ and $dtt^*\phi_1$, but only one will actually be chosen. The walk will then continue to investigate the appropriate argument.
- If ϕ is a walk arriving from either the true ϕ_2 or the false ϕ_3 branch (so the answer to the result of the conditional) then the walk is extended out of the conditional at the top with $d^*\phi_2$ or $d^*\phi_3$.

Recursion

We will adopt the coding that we suggested in Chapter 3 which will encode recursion in terms of contraction and *application*. Hence all the possible walks through recursion have already been defined in terms of the multiplicative and exponential connectives.

We now need to extend the dynamic algebra with the constants that we require. We will again overload notation and use the same names for the labels and the algebra.

Definition 4.2.11 (Λ_{pcf}^*)

We define the algebra Λ_{pcf} with the following constants which are the labels that we have already seen:

- Natural numbers n , which we shall write as $\bar{n}$ to avoid confusion with the constants 0 and 1; and
- Boolean constants $\mathbf{tt}, \mathbf{tt}^*$ and $\mathbf{ff}, \mathbf{ff}^*$, together with
- Function constants $\mathbf{succ}, \mathbf{succ}^*, \mathbf{pred}, \mathbf{pred}^*$ and $\mathbf{iszero}, \mathbf{iszero}^*$.

There are corresponding annihilation equations for these constants which are essentially the δ -rules for PCF:

$$\begin{array}{llll}
 \mathbf{tt}^* \mathbf{tt} = \mathbf{ff}^* \mathbf{ff} & = & 1 & \mathbf{ff}^* \mathbf{tt} = \mathbf{tt}^* \mathbf{ff} & = & 0 \\
 \mathbf{iszero}^* \bar{0} \mathbf{iszero} & = & \mathbf{tt} & \mathbf{iszero}^* \overline{n+1} \mathbf{iszero} & = & \mathbf{ff} \\
 \mathbf{pred}^* \bar{0} \mathbf{pred} & = & \bar{0} & \mathbf{pred}^* \overline{n+1} \mathbf{pred} & = & \bar{n} \\
 \mathbf{succ}^* \bar{n} \mathbf{succ} & = & \overline{n+1} & & &
 \end{array}$$

Where 0 is an absorbing element, and 1 is the identity. As before, we have omitted to write the *multiplication* operation.

The intuition is that f together with f^* give the corresponding interpretation of the PCF arithmetic functions.

We now define the algebra Λ_{pcf}^* to be the product of the algebras Λ^* and Λ_{pcf} . $\diamond$

It should be noted that there is *no* interaction between the equations for the PCF constants and Λ^* ; they are operating on disjoint domains. However, we are going to abuse notation considerably and write the elements of the product of the algebra together.

Remark 4.2.12 In the same way as one can read p (resp. q) as *push* 0 (resp. *push* 1), and p^* (resp. q^*) as *pop* 0 (resp. *pop* 1) [Gir89a] we have an analogy for the PCF constants. Hence, $\mathbf{tt}$ is *push true* and $\mathbf{tt}^*$ is *pop true*, and similarly for $\mathbf{ff}$. $\diamond$

There is a very simple model that shows how we can compute in this algebra.

Definition 4.2.13 Let *rand* and *rator* be *operand* and *operator* stacks respectively. We define the action of the PCF elements of Λ_{pcf}^* on the pair $(\text{rand}, \text{rator})$ as follows:

$\text{tt}(\text{rand}, \text{rator})$	$= (\text{tt} : \text{rand}, \text{rator})$
$\text{ff}(\text{rand}, \text{rator})$	$= (\text{ff} : \text{rand}, \text{rator})$
$\text{tt}^*(\text{tt} : \text{rand}, \text{rator})$	$= (\text{rand}, \text{rator})$
$\text{ff}^*(\text{ff} : \text{rand}, \text{rator})$	$= (\text{rand}, \text{rator})$
$\bar{n}(\text{rand}, \text{rator})$	$= (n : \text{rand}, \text{rator})$
$\text{succ}(\text{rand}, \text{rator})$	$= (\text{rand}, \text{succ} : \text{rator})$
$\text{succ}^*(n : \text{rand}, \text{succ} : \text{rator})$	$= (n + 1 : \text{rand}, \text{rator})$
$\text{pred}(\text{rand}, \text{rator})$	$= (\text{rand}, \text{pred} : \text{rator})$
$\text{pred}^*(0 : \text{rand}, \text{pred} : \text{rator})$	$= (0 : \text{rand}, \text{rator})$
$\text{pred}^*(n + 1 : \text{rand}, \text{pred} : \text{rator})$	$= (n : \text{rand}, \text{rator})$
$\text{iszero}(\text{rand}, \text{rator})$	$= (\text{rand}, \text{iszero} : \text{rator})$
$\text{iszero}^*(0 : \text{rand}, \text{iszero} : \text{rator})$	$= (\text{tt} : \text{rand}, \text{rator})$
$\text{iszero}^*(n + 1 : \text{rand}, \text{iszero} : \text{rator})$	$= (\text{ff} : \text{rand}, \text{rator})$

◇

Proposition 4.2.14 The above definition yields a sound model of the equational theory for the PCF elements of Λ_{pcf}^* . ◇

PROOF: Straightforward analysis of the equations. □

Definition 4.2.15 (PCF paths)

A path ϕ in a PCF proof net is a walk such that $\Lambda_{\text{pcf}}^* \vdash \phi \neq 0$.

The notion of an execution path for Λ_{pcf}^* is exactly the same as before. ◇

Remark 4.2.16 By definition of a path in a PCF structure all occurrences of arithmetic functions are balanced, i.e. we never have path constructions of the form $\text{iszero}^* \bar{n} \text{succ}$, for example. ◇

Example 4.2.17 We now show two simple examples to show how things work. Here we are assuming the “ $!(A \multimap B)$ ” translation of PCF into proof structures given in Chapter 3.

1. The first example is very simple and included to give a feel of how the theory works. Consider the term: $(\lambda x. \text{succ } x)3$. Starting at the root, we build the (unique!) path which yields the following:

$$q^* d^* !(q ! (\text{succ}^* d^*) t^* p^*) dp ! (\bar{3}) p^* d^* !(pt ! (dsucc) q^*) dq$$

We show a few snap-shots of the reduction:

- $q^*q!(\text{succ}^*d^*)t^*p^*p!(\overline{3})p^*pt!(d\text{succ})q^*q$
- $!(\text{succ}^*d^*)t^*(\overline{3})t!(d\text{succ})$
- $!(\text{succ}^*d^*)t^*t!(\overline{3})!(d\text{succ})$
- $!(\text{succ}^*d^*)(\overline{3})!(d\text{succ})$
- $!(\text{succ}^*d^*(\overline{3})d\text{succ})$
- $!(\text{succ}^*d^*\overline{3}\text{succ})$
- $!(\text{succ}^*\overline{3}\text{succ})$

which reduces to $!(\overline{4})$ as required.

2. The second example is included to show how the conditional works.

$$(\lambda x.\text{cond } x \text{ ff tt})(\text{iszero } \overline{2})$$

There is a single unique path starting at the root:

$$q^*d^*(q!(d^*)t^*(\text{tt})t!(d\text{ff}^*d^*)t^*p^*)dp!(\text{iszero}^*d^*)t^*(\overline{2})t!(d\text{iszero})p^*d^*(pt!(d)q^*)dq$$

Again, we show a few snap-shots of the reduction:

- $q^*q!(d^*)t^*(\text{tt})t!(d\text{ff}^*d^*)t^*p^*p!(\text{iszero}^*d^*)t^*(\overline{2})t!(d\text{iszero})p^*pt!(d)q^*q$
- $!(d^*)t^*(\text{tt})t!(d\text{ff}^*d^*)t^*(\text{iszero}^*d^*)t^*(\overline{2})t!(d\text{iszero})t!(d)$
- $!(d^*)t^*t!(\text{tt})!(d\text{ff}^*d^*)t^*(\text{iszero}^*d^*)t^*t!(\overline{2})t!(d\text{iszero})!(d)$
- $!(d^*)!(\text{tt})!(d\text{ff}^*d^*)t^*t!(\text{iszero}^*d^*)!!(\overline{2})!!(d\text{iszero})!(d)$
- $!(d^*)!(\text{tt})!(d\text{ff}^*d^*)!(\text{iszero}^*d^*)(d)!(d)(\overline{2}\text{iszero})$
- $!(d^*)!(\text{tt})!(d\text{ff}^*d^*)(\text{iszero}^*d^*)(\overline{2}\text{iszero})$
- $!(d^*)!(\text{tt})d\text{ff}^*(\text{iszero}^*\overline{2}\text{iszero})$
- $!(d^*)(d)!(\text{ttff}^*)(\text{ff})$
- $!(\text{ttff}^*\text{ff})$

which reduces to $!(\text{tt})$ as required.

◇

4.3 Context Semantics

Here we give our presentation of a context semantics, and show that context trees give a model of the dynamic algebra Λ^* . We go a little further in fact by totally separating out the multiplicative and exponential information.

Definition 4.3.1 A context is a pair $(\mathcal{M}, \mathcal{E})$, where:

$\mathcal{M}$ is a multiplicative list:

$$\mathcal{M} ::= l : \mathcal{M} \mid r : \mathcal{M} \mid \square$$

$\mathcal{E}$ is an exponential tree:

$$\mathcal{E} ::= L : \mathcal{E} \mid R : \mathcal{E} \mid \langle \mathcal{E}, \mathcal{E} \rangle \mid \square$$

Where “:” is a right associative cons operation, and $\square$ is equally an empty context tree and an empty list. $\diamond$

The purpose of a context semantics is to provide a model of the dynamic algebra Λ^* . The multiplicative component of the context is simply a flat list which records the sequence of p and q operations. The exponential context trees are considerably more complicated. The general shape of a tree is $\langle A, \langle B, \langle C, \dots \rangle \rangle \rangle$, where A, B , and C are context trees. The intuition to hold onto is that the depth of a context indicates the level of nesting of the exponentials in Linear Logic.

We now define the following operations on the $(\mathcal{M}, \mathcal{E})$ data structure using the elements of the dynamic algebra. Hence operations are context transformers.

As with the small model, (Definition 4.2.5), we define $\llbracket 1 \rrbracket$ as the identity on contexts, and $\llbracket 0 \rrbracket$ is the nowhere defined context transformer. $\llbracket f^* \rrbracket$ is the inverse transformation $\llbracket f \rrbracket^{-1}$. Composition is lifted to the model in the natural way: $\llbracket ab \rrbracket m = \llbracket a \rrbracket (\llbracket b \rrbracket m)$.

Multiplicatives

$$\llbracket p \rrbracket(m, e) = (l : m, e) \quad \llbracket q \rrbracket(m, e) = (r : m, e)$$

Exponentials

$$\begin{aligned} \llbracket r \rrbracket(m, e) &= (m, L : e) & \llbracket s \rrbracket(m, e) &= (m, R : e) \\ \llbracket t \rrbracket(m, \langle e_1, \langle e_2, e_3 \rangle \rangle) &= (m, \langle \langle e_1, e_2 \rangle, e_3 \rangle) & \llbracket d \rrbracket(m, e) &= (m, \langle \square, e \rangle) \end{aligned}$$

All these work together with an exponential morphism !:

$$\begin{aligned} \llbracket !(f) \rrbracket(m, \langle e_1, e_2 \rangle) &= \text{let } (m', e'_2) = \llbracket f \rrbracket(m, e_2) \\ &\quad \text{in } (m', \langle e_1, e'_2 \rangle) \end{aligned}$$

Proposition 4.3.2 The above operations on $(\mathcal{M}, \mathcal{E})$ give a sound model of the algebra Λ^* . $\diamond$

PROOF: We show that all the equations given in Definition 4.2.4 hold in the model: If $\Lambda^* \vdash a = b$ then $(\mathcal{M}, \mathcal{E}) \models \llbracket a \rrbracket(c) = \llbracket b \rrbracket(c)$. We verify just a few of the equations, and we will drop the semantic brackets.

$$\begin{aligned} p^*p(m, e) &= p^*(l : m, e) & !(f)d(m, e) &= !(f)(m, \langle \square, e \rangle) \\ &= (m, e) & &= \text{let } (m', e') = f(m, e) \\ & & &\quad \text{in } (m', \langle \square, e' \rangle) \\ & & &= \text{let } (m', e') = f(m, e) \\ & & &\quad \text{in } d(m', e') \\ & & &= df(m, e) \end{aligned}$$

$$\begin{aligned} !(f)!(g)(m, \langle e_1, e_2 \rangle) &= \text{let } (m', e'_2) = g(m, e_2) \\ &\quad \text{in } !(f)(m', \langle e_1, e'_2 \rangle) \\ &= \text{let } (m', e'_2) = g(m, e_2) \\ &\quad \text{in let } (m'', e''_2) = f(m', e'_2) \\ &\quad \text{in } (m'', \langle e_1, e''_2 \rangle) \\ &= \text{let } (m'', e''_2) = fg(m, e_2) \\ &\quad \text{in } (m'', \langle e_1, e''_2 \rangle) \\ &= !(fg)(m, \langle e_1, e_2 \rangle) \end{aligned}$$

$$\begin{aligned} !(f)t(m, \langle e_1, \langle e_2, e_3 \rangle \rangle) &= !(f)(m, \langle \langle e_1, e_2 \rangle, e_3 \rangle) \\ &= \text{let } (m, e'_3) = f(m, e_3) \\ &\quad \text{in } (m', \langle \langle e_1, e_2 \rangle, e'_3 \rangle) \\ &= \text{let } (m, e'_3) = f(m, e_3) \\ &\quad \text{in } t(m', \langle e_1, \langle e_2, e'_3 \rangle \rangle) \\ &= t!(f)(m, \langle e_1, \langle e_2, e_3 \rangle \rangle) \end{aligned}$$

□

The reason for the separation of the context semantics into multiplicative and exponential information is two-fold:

- We feel that the decomposition provides a clearer understanding of the models of this theory; basically we want to separate out two very independent entities.
- In Chapter 6 we look at implementing directly the context semantics as a data-flow machine. This decomposition provides a substantial speedup of this implementation.

Contexts for PCF

We set up a context semantics for PCF which will be identical to the context semantics used thus far, except there is an additional piece of information that we need to carry around.

With respect to Remark 4.2.16 we can model the constants on *one* stack, eliminating the operator stack. More precisely we can drop the arithmetic functions f , and keep only f^* . Additionally we drop the $-^*$ notation for these arithmetic functions.

We will consider contexts now to be triples: $(\mathcal{M}, \mathcal{E}, \mathcal{D})$ where $\mathcal{M}$ and $\mathcal{E}$ are as before, and we define $\mathcal{D}$ as:

$$\mathcal{D} ::= n \mid \mathbf{tt} \mid \mathbf{ff} \mid \square$$

The intuition is that $\square$ is an empty datum, and the remaining are the possible (partial) results that can be returned from a computation. Note that we again overload notation and write elements of a context using the same names as the syntax of PCF and Λ_{pcf}^* .

Remark 4.3.3 Contexts seem to correspond to moves in PCF games of [AJM94]. $\square$ is a *question* and other values are *answers* in the notation of Games. Hence one can see contexts as *moves* in a Game; for example a context $(m, e, \square)$ is a question, and (m, e, d) is an answer. $\diamond$

The following defines the action of the PCF elements of Λ_{pcf}^* in this data structure:

Definition 4.3.4

$$\begin{array}{llll} \mathbf{tt}(m, e, \square) & = & (m, e, \mathbf{tt}) & \mathbf{ff}(m, e, \square) & = & (m, e, \mathbf{ff}) \\ \mathbf{tt}^*(m, e, \mathbf{tt}) & = & (m, e, \square) & \mathbf{ff}^*(m, e, \mathbf{ff}) & = & (m, e, \square) \\ \mathbf{iszero}(m, e, 0) & = & (m, e, \mathbf{tt}) & \mathbf{iszero}(m, e, n+1) & = & (m, e, \mathbf{ff}) \\ \mathbf{pred}(m, e, 0) & = & (m, e, 0) & \mathbf{pred}(m, e, n+1) & = & (m, e, n) \\ n(m, e, \square) & = & (m, e, n) & \mathbf{succ}(m, e, n) & = & (m, e, n+1) \end{array}$$

◇

Proposition 4.3.5 The above operations on $(\mathcal{M}, \mathcal{E}, \mathcal{D})$ give a sound model of the algebra Λ_{pcf}^* . ◇

PROOF: The proof proceeds in the same way as the proof of proposition 4.2.6, with the addition of the PCF constants. Here we just show these constants and verify the equations given in Definition 4.2.11.

$$\begin{aligned} \mathbf{tt}^* \mathbf{tt}(m, e, \square) &= \mathbf{tt}^*(m, e, \mathbf{tt}) & \mathbf{succ} \, \bar{n}(m, e, \square) &= \mathbf{succ} \, (m, e, n) \\ &= (m, e, \square) & &= (m, e, n+1) \\ & & &= \overline{n+1}(m, e, \square) \end{aligned}$$

□

For the purpose of our work the property of soundness is sufficient – all we require is that this model is good enough to model computations. In Chapter 6 we will provide an implementation based on performing computations using PCF contexts.

4.4 Other Notions of Paths

In this section we will review several notions of a path in a λ -term; more specifically we will talk about paths in Linear Logic proof structures from which we obtain paths in the λ -calculus from the translations in Chapter 3.

In the literature there are several notions of paths that provide a different perspective on the same theory.

- A *persistent* path concerns the notion of a path surviving the action of a cut. This has been studied by Danos and Regnier [DR94], and gives a *geometrical* approach to paths.
- A *legal* path. This has been studied by Asperti and Laneve [AL93b], and gives a *parentheses matching* approach to paths.
- A *consistent* path is defined in terms of *context semantics* — a labelling of context marks on a proof structure. This has been studied by Gonthier, Abadi and Lévy [GAL92], and gives a *token pushing* approach to paths.
- Finally, we believe that the strategies of PCF games [AJM94] provide an alternative view on the same theory.

We recommend very strongly the results in [ADLR94], where several notions of paths are shown to be equivalent.

Much work on paths in Linear Logic proof structures has been done by Danos [Dan90] and Regnier [Reg92]. A very readable account of the relationship between Lévy labels and paths is given in [Lan93] and also [AL93b]. We refer the reader to the original works for a more complete presentation.

4.5 Dynamic Algebras

In this section we look at some basic properties of the dynamic algebra Λ_{pcf}^* that we are using; keeping in the spirit of splitting the presentation into multiplicative, exponential and data.

Multiplicatives

We begin by looking at some properties of the multiplicative fragment.

Definition 4.5.1 We say that a term $t \in \Lambda^*$ is *positive* if it contains no sub-term of the form u^* . A term is *negative* if it only contains elements of the form u^* . $\diamond$

One sees a path as negative if it is moving into a term, and positive if it is moving out of a term.

Proposition 4.5.2 If ϕ is a positive monomial, then $\phi^*\phi = 1$. $\diamond$

PROOF: The proof proceeds by an induction on the length of ϕ . If ϕ is an atomic element p, q or 1 , then the result follows trivially by the equations of the theory. If ϕ is term $t_1 \cdots t_n$ then $\phi^*\phi = t_n^* \cdots t_1^* t_1 \cdots t_n$. Now $t_1^* t_1 = 1$ from the equational theory, and the process of annihilation will continue until $\phi^*\phi = 1$. $\square$

Definition 4.5.3 A term $t \in \Lambda^*$ is in *stable* (or AB^*) form if it has one of the following shapes:

1. 0 or 1 .
2. $t = t_1 \cdots t_n$ such that for some $i : 0 \leq i \leq n$ $t_1 \cdots t_i$ is *positive*, and $t_{i+1} \cdots t_n$ is *negative*.

$\diamond$

Theorem 4.5.4 Any multiplicative term of Λ^* can be written in stable form.

PROOF: We show this by induction on the structure of $t \in \Lambda^*$.

The cases when $t \equiv 0, 1, p$ or q are trivially in this form.

There are three inductive cases:

- If $t \equiv u^*$, then by hypothesis, $u = a_1 \dots a_i b_{i+1} \dots b_m$ is in stable form where $a_1 \dots a_i$ are positive and $b_{i+1} \dots b_m$ are negative. Now, $u^* = b_m^* \dots b_{i+1}^* a_i^* \dots a_1^*$ which is again in stable form.
 - If $t \equiv t_1 t_2$, then by hypothesis t_1 and t_2 are in stable form. The only interesting case to consider is when $t_1 = a_1 \dots a_i b_{i+1} \dots b_m$ and $t_2 = a'_1 \dots a'_j b'_{j+1} \dots b'_n$, and $b_m a'_1$ is either $p^* p$, $q^* q$, $q^* p$ or $p^* q$. The last two cases reduce the term t to 0 which is in stable form. The first two cases reduces $t_1 t_2$ to $a_1 \dots a_i b_{i+1} \dots b_{m-1} a'_2 \dots a'_j b'_{j+1} \dots b'_n$.
- This process will continue until a stable form is reached. The intuition is that the negative terms move to the right, and the positive terms move to the left.
- If $t \equiv t_1 + t_2$ then the result is immediate by application of the hypothesis twice.

□

It is worth pausing for a moment to look at what this theory can give us. We have the linear λ -calculus represented as a set of paths. The paths embody *all* the information about the term, including the notion of substitution—free from the notion of syntax in the λ -calculus. The above result gives a notion of Strong Normalisation and the system is Church-Rosser.

Finally, we describe the global structure of paths in a multiplicative proof structure. Consider the multiplicative cut of par against a tensor. If we arrive from the hypothesis of the tensor, along, say, q then we have to enter the q^* premise of the par. The path in the proof above par can either lead to the path returning through q or p . If we return through q , then we return to the q^* premise of the tensor. Hence we have returned to the exact point from where we started. If, however, we returned from the p side of the par, then we have to return up the p^* premise of the tensor. Since there is no connection between the premises of a tensor, we are obliged to return through p , and hence return through p^* of the par node.

By repeated application of this idea, we get the notion of paths that return on themselves². Moreover, the structure of this path is well behaved in the the so-called *well bracketed* discipline. This is exactly the notion of well balanced path as studied by Asperti and Laneve [AL93b].

²Called *epistrophique* by Danos [Dan90].

Exponentials

We now look at the exponentials, and try to extend the above sequence of results. For the multiplicative fragment, the above results apply to *all* paths, not necessarily those arising from λ -terms. For the exponentials however, we are interested in paths arising from a particular translation. This arises by the fact that the different translations from the λ -calculus differ only in where we place the exponential information.

First, recall that we are dealing with the $D =!(D \multimap D)$ translation of the λ -calculus into Linear Logic proof structures. This gives us a particular shape of paths in λ -terms which we state below in the stable form theorem. Note that for the other translation $D =!D \multimap D$ the shape of stable forms is given by $\phi ::= A!(\phi)B^* \mid 1$. This is reported in [Reg92].

Definition 4.5.5 A non-zero exponential term t is in stable (AB^*) form if it has the following shape $\phi ::= !(A\phi B^*) \mid 1$. $\diamond$

Theorem 4.5.6 Any path in a proof structure arising from the $D =!(D \multimap D)$ translation can be written in stable form.

PCF constants

We now look at some properties of execution paths for PCF terms. To motivate the forthcoming result, we show an example.

Example 4.5.7 Consider again the PCF term $(\lambda x.\text{succ } x)3$ from Example 4.2.17. We show first the execution path, then the sequence of context transformations induced. We will identify the questions and answers, and the associated bracketing condition.

The execution path is:

$$q^*d^*!(q!(\text{succ}d^*)t^*p^*)dp!(\overline{3})p^*d^*!(pt!(d)q^*)dq$$

If we examine the structure of this term carefully, we can write it as:

$$\phi_1^*!(\text{succ})\phi_2^*(\overline{3})\phi_2\phi_1$$

where $\phi_1 =!(q^*)dq$ and $\phi_2 = p^*d^*!(pt!(d))$.

ϕ_1 is the path from the root to the **succ**, and ϕ_2 is the path from the **succ** to the argument $\overline{3}$. ϕ_2^* and ϕ_1^* are the return paths respectively.

This throws up a very interesting phenomenon that the author prefers to think of as a question and answer discipline of these paths. The scenario is that we start at the root asking a question, say q_1 . When the path arrives at the **succ**, we start asking a new question q_2 which is the argument to the successor. The path continues until it reaches the constant $\bar{3}$ at which point it answers the previously asked question. The path returns along the same path backwards to the place where the question was asked (the **succ**) which completes answering the question q_2 . The **succ** is then applied, and the path then continues back to the root where the original question q_1 was asked.

(The reader may find this example clearer by using the **succ**^{*}, **succ** notation.) $\diamond$

For programs of base type we have a stronger result:

Definition 4.5.8 (Well Balanced Paths)

A well-balanced path v in Λ_{pcf}^* is built from the following grammar:

$$\begin{array}{lcl} v & ::= & \phi^* v \phi \\ & | & f \ v \\ & | & vv \\ & | & \bar{n} \mid \mathbf{tt} \mid \mathbf{tt}^* \mid \mathbf{ff} \mid \mathbf{ff}^* \mid 1 \end{array}$$

where ϕ is a term of Λ^* .

$\diamond$

Proposition 4.5.9 All maximal paths in Λ_{pcf}^* arising from correct PCF proof net structures are well-balanced paths. $\diamond$

PROOF: The proof proceeds by an induction over the structure of PCF terms M . We use the result of Asperti and Laneve [AL93b] that paths are well balanced for the λ -calculus, hence we just need to verify the cases for the PCF constants.

It is trivial to verify the proposition holds for the constants $M = \bar{n}, \mathbf{tt}$ and $\mathbf{ff}$ since the path construction is given by the corresponding term in Λ_{pcf}^* .

If M is the application fM' of some arithmetic function $f \in \{\mathbf{succ}, \mathbf{pred}, \mathbf{iszero}\}$ then by the hypothesis M' is a well balanced path v and the result follows by the path construction fv .

If M is the application of a conditional **cond** $M_1 \ M_2 \ M_3$, then by application of the hypothesis three times there are well balanced paths v_1, v_2 and v_3 . The path construction of the conditional is given by $d^*v_2d\mathbf{tt}^*v_1$ if $\Lambda_{\text{pcf}}^* \vdash v_1 = \mathbf{tt}$, or $d^*v_3d\mathbf{ff}^*v_1$ if $\Lambda_{\text{pcf}}^* \vdash v_1 = \mathbf{ff}$. $\square$

Moreover, we have that if $\phi^*\phi$ is defined, then $\phi^*\phi = 1$ (so there is a context C such that $\phi^*\phi C = C$). This gives the final result of this section:

Proposition 4.5.10 Let M be a PCF term at ground type B , then for the execution path ϕ , $\Lambda_{\text{pcf}}^* \vdash \phi = c$ where c is the interpretation of the constant of type B . $\diamond$

PROOF: (Sketch) Since the only paths in a proof structure are the “good ones”, and these survive the action of cut-elimination, it suffices to talk about terms in normal form. The result follows in a straightforward way by the interpretation of the constants. $\square$

The notion that “paths return on themselves” in a well balanced discipline will be used in Chapter 6 where we suggest some optimisations on the implementation.

4.5.1 A Path Semantics for PCF

We have shown thus far how to give a path construction for the PCF constants and Linear Logic proof structures. Using the translations of Chapter 3, this yields a path semantics for full PCF.

For a term of base type, we have the result that a path is a monomial, that has a weight equal to the value of the normal form of the term. However, for terms that do not have base type normal forms we have found a tree representation of the set of paths useful. The tree structure enumerates the monomials in the term in a form that resembles the term itself.

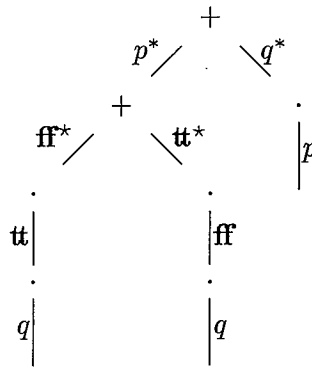
We can write the set of paths associated with a PCF-term as a *tree* of paths defined below.

Definition 4.5.11 (PCF trees)

A *path tree* associated to a polynomial in Λ_{pcf}^* is a binary tree with edges labelled by elements of the algebra, and nodes labelled by either composition $\cdot$ or sum $+$. $\diamond$

Example 4.5.12 We show the PCF term for negation, without exponentials.

$\lambda x.\text{cond } x \text{ ff } \text{tt}$



$\diamond$

4.6 Relating Paths, Labels and Types

In this section we will briefly mention, in the form of examples, some connections between paths, labels and types to give the reader a wider perspective on this theory. We begin with the connection between Paths and Labels, in particular, we will show how Lévy labels relate strongly to the multiplicative elements of Λ_{pcf}^* .

By the context semantics a sequence of multiplicative context transformers induces a *trace* of the computation—we can associate a context for each edge on the graph as we traverse the structure, and a context transformer between each edge. Alternatively, we can use Lévy labels as edges, with the atomic letters as names for them.

The following definition shows how we can move from labels of reduced terms to elements of Λ_{pcf}^* .

Definition 4.6.1

$$\begin{aligned} h(a) &= 1 \text{ (atomic } a) \\ h(\alpha\beta) &= h(\beta) \cdot h(\alpha) \\ h(\overline{\alpha}) &= q^* \cdot h(\alpha) \cdot q \\ h(\underline{\alpha}) &= p^* \cdot h(\alpha)^r \cdot p \end{aligned}$$

where ϕ^r , the reverse path is defined in the obvious way:

$$\begin{aligned} c^r &= c \text{ for } c \in \{1, p, q\} \\ (\phi^*)^r &= (\phi^r)^* \\ (\phi_1\phi_2)^r &= \phi_2^r\phi_1^r \end{aligned}$$

i.e. the letters give names to the label (context), but it is the underline/overline structure that gives the information.

Note in particular that h reverses the path since labels are read from left to right, and paths from right to left. $\diamond$

Example 4.6.2 We will use a very simple example to show how things work. Consider the labelled reduction of $(\lambda x.x)\mathbf{tt}$.

- $((\lambda x.x^a)^b\mathbf{tt}^c)^d$
- $\mathbf{tt}^{d\overline{b}a\underline{b}c}$

Applying the function h to this label yields a path: $h(d\overline{b}a\underline{b}c) = p^*pq^*q$.

The execution path for this term is given by: $q^*qp^*p\mathbf{tt} \ p^*pq^*q$, with the obvious connection with the labelling.

Hence an alternative notion of a path in a λ -term is given by reducing a labelled λ -term, keeping the history of the computation in the label. This history is *exactly* the path the token must traverse in the original network. Hence we have the following very tight correspondence — computing the path = reducing the term. $\diamond$

Definition 4.6.3

There is an evident notion of nesting *depth* of a label. The maximum depth of a labelled term can be given by:

$$\begin{aligned} h(a) &= 0 \text{ (atomic } a) \\ h(\alpha\beta) &= \max(h(\alpha), h(\beta)) \\ h(\bar{\alpha}) &= 1 + h(\alpha) \\ h(\underline{\alpha}) &= 1 + h(\alpha) \end{aligned}$$

$\diamond$

Hence the labels give a bound required for the multiplicative stack.

We now show the connection between labels/paths and types. This work has come from the observation of Lévy in the appendix of [Lev78], which our first definition comes from.

Definition 4.6.4 Types can be extracted from labels as follows. Let $\mathcal{T}$ be a set of types $\{A \rightarrow A, A \rightarrow A \rightarrow A, (A \rightarrow A) \rightarrow A \dots\}$, where A is an *atomic* type, then we define a homomorphism on labels as follows:

$$\begin{aligned} h(a) &\in \mathcal{T} \text{ (atomic } a) \\ h(\alpha\beta) &= h(\beta) \\ h(\bar{\alpha}) &= v \text{ if } h(\alpha) = u \rightarrow v \\ h(\underline{\alpha}) &= u \text{ if } h(\alpha) = u \rightarrow v \end{aligned}$$

The restriction on A being an atomic type means that we are working with η -expanded terms, i.e. Axiom links are at atomic types. $\diamond$

Since the nesting of labels relates to both the type and the size of the multiplicative context stack, there is a trivial observation that the size of the type gives a bound on the size of the multiplicative stack. In fact we can see that nested redexes (cuts) give an exponential growth in the length of the path. For example $((((II)I)I)I)$ (left spines) has exponential growth, whereas $I(I(I(II)))$ (right spines) has a linear growth.

Example 4.6.5

We show the labelled reduction for these two terms. For $((II)I)I$ we have the following sequence of reductions:

- $(((((\lambda x.x^a)^b(\lambda x.x^c)^d)(\lambda x.x^f)^g)^h(\lambda x.x^i)^j)^k$
- $(\lambda x.x^i)^{\overline{khe\bar{b}ab\bar{d}c\bar{e}\bar{b}ab\bar{d}g}f\overline{he\bar{b}ab\bar{d}c\bar{e}\bar{b}ab\bar{d}g}j}$

For $I(I(II))$ we have the following sequence of reductions:

- $((\lambda x.x^a)^b((\lambda x.x^c)^d((\lambda x.x^e)^f(\lambda x.x^g)^h)^i)^j)^k$
- $((\lambda x.x^c)^d((\lambda x.x^e)^f(\lambda x.x^g)^h)^i)^{k\bar{b}a\bar{b}j}$
- $((\lambda x.x^e)^f(\lambda x.x^g)^h)^{k\bar{b}a\bar{b}j\bar{d}c\bar{d}i}$
- $(\lambda x.x^g)^{k\bar{b}a\bar{b}j\bar{d}c\bar{d}i\bar{f}e\bar{f}h}$

◇

This example suggests that the length of the label is related to the *size* of the type. In fact the definition of the homomorphism of labels to types also gives this evidence. Another interesting example to show the path increasing exponentially is $\Delta\Delta$ where $\Delta = (\lambda d.dd)$. The first few labelled reductions are as follows:

1. $((\lambda d.(d^a d^b)^c)^d(\lambda d.(d^e d^f)^g)^h)^i$
2. $((\lambda d.(d^e d^f)^g)^{a\bar{d}h}(\lambda d.(d^e d^f)^g)^{b\bar{d}h})^{i\bar{d}c}$
3. $((\lambda d.(d^e d^f)^g)^{e\bar{a}d\bar{h}b\bar{d}h}(\lambda d.(d^e d^f)^g)^{f\bar{a}d\bar{h}b\bar{d}h})^{i\bar{d}c\bar{a}\bar{d}\bar{h}g}$
4. $((\lambda d.(d^e d^f)^g)^{ee\bar{a}d\bar{h}b\bar{d}hf\bar{a}d\bar{h}b\bar{d}h}(\lambda d.(d^e d^f)^g)^{fe\bar{a}d\bar{h}b\bar{d}hf\bar{a}d\bar{h}b\bar{d}h})^{i\bar{d}c\bar{a}\bar{d}\bar{h}ge\bar{a}d\bar{h}b\bar{d}hg}$

Note that the length of the outermost label doubles at every step of reduction. This example also shows a curious phenomenon that the execution path of $\Delta\Delta$ generates the natural numbers. We leave the reader to work out how to extract these (binary) numbers from the labels in the above term. This example is also reported in [Dan90] for execution paths in Λ^* .

4.7 Discussion

In this chapter we have seen how to construct a set of paths for PCF terms in the dynamic algebra Λ_{pcf}^* . This style of semantics is sufficient to represent the notion of computation, and this will now be used for the purposes of proving the correctness of an Interaction net implementation in the next chapter, and in the following chapter, actually used as an implementation technique itself.

Results of this chapter will be used to justify some of the optimisations that we will perform on our implementation of the Geometry of Interaction Machine.

Our interests here have not been to analyse the reduction strategy for PCF in terms of the algebra Λ_{pcf}^* ; we were using PCF purely as a convenient well-known syntax. Further work of this connection is out of the scope of this thesis, but we feel that there are results to be obtained.

There is a striking similarity between the properties of paths for PCF terms and the Games model [AJM94]. We hinted at this but must leave a detailed treatment for further work.

Paths for general recursive data structures is an obvious next step to this work which will give a semantics and implementation paradigm for larger, more realistic functional programming languages. We will hint at some of the ideas in Chapter 6 where we will show how to implement some of these concepts.

Chapter 5

A Lambda Evaluator Based on Interaction Nets

This chapter gives the first major contribution to the thesis. We present an interaction net implementation for the λ -calculus based in Lafont's Interaction Nets. We compare this with existing implementations, and also suggest further extensions.

5.1 Introduction

In this chapter we will provide a very simple implementation of the λ -calculus using ideas that have been used in the implementation of optimal reduction. Namely, we will use Lafont's Interaction Nets, and the idea of using fan-in and fan-out nodes as described in Chapter 2.

The emphasis of this work will *not* be on achieving optimal reduction itself—a criterion based on number of β -reductions—but rather we base our criterion on performing the least amount of work for *practical* programs; a notion to be made a little more precise later.

Lamping [Lam90] and Kathail [Kat90] independently discovered the techniques required to implement optimal reduction. Although neither of these algorithms were based on *interaction nets*, the former was used as a basis for an interaction net optimal reducer reported in [GAL92] where Lamping's algorithm was considerably cleaned up using insight gained from Linear Logic. Since the work of [GAL92] seems to be the most studied and documented of techniques, we will use it as a basis for making comparisons to our work.

In that paper the most significant features are:

- an Interaction Net formalism;

- complete reductions in the λ -calculus—terminating in full normal form;
- optimal reduction as defined by Lévy.
- no Garbage Collection;
- Linear Logic foundation; allowing the dynamic semantics, discussed in Chapter 4, to be used as a proof technique.

A little analysis of this algorithm soon indicates a tradeoff: optimality versus bookkeeping. To quote Lamping [Lam90]:

One discouraging note: ...both an informal argument and experience with an implementation of the algorithm indicate that the amount of bookkeeping work the algorithm requires for each β -reduction step is proportional to the cost of doing a substitution ...

It has also been reported [AD94] that one of the main problems of these optimal reduction implementations is the accumulation of nodes which should really cancel each other out; see end of Section 4.2. The restriction on an interaction net however precludes such a scheme of adding additional annihilation equations. In *op. cit.*, an implementation is considered which greatly increases the performance of these algorithms, but takes us away from an interaction net paradigm.

The focus of these works has been on achieving optimal reduction. Consequently this has led to an implementation with a very complicated data structure and rewriting system. Our work will address the problems of using this kind of framework, (interaction nets), for implementing functional programming languages. The aim will be to try to keep the data structures and dynamics simple, and we will be prepared to trade off some of the bookkeeping for optimality.

Our focus will be on issues related to a pure functional language (PCF) and we only hint at some additional features to support more sophisticated languages. One of the most striking features listed above which will be of concern to an implementation is the issue of *complete reductions*. It is common knowledge that a weak notion of reduction is the current trend and we adhere to this. Our main result of this chapter is an interaction net formalism for implementing the λ -calculus, terminating before full normal form. This is achieved by a restricted form of substitution; at the crucial point where the extant algorithms pay heavily. We will discuss performance issues at the end of the chapter.

To summarise, we propose an interaction net implementation for PCF, with the following features:

- Reduction mimics cut elimination of Linear Logic very closely, allowing a simple correctness proof, and an easier understanding of the algorithm.
- We use some of the techniques from the work on optimal reduction to allow partial sharing.

5.2 Interaction Nets

In [Laf90, Laf92] Lafont describes a new paradigm in programming languages and implementation based on what he calls *interaction nets*; a networked system of interacting agents founded on proof nets for Linear Logic.

These nets are very appealing from a computational point of view. On the one hand we have a very simple, graphical rewriting system which enjoys properties such as confluence, and on the other hand there are trivial parallel implementations.

We briefly review the paradigm.

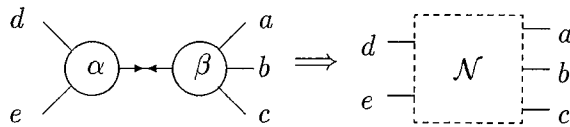
Definition 5.2.1 (Interaction Net)

An interaction net is specified by the following data.

- A set Σ of symbols (nodes), each with an arity $p \in \mathbb{N}$. A net on Σ is an undirected graph with vertices labelled by symbols $\alpha, \beta \in \Sigma$. p is the number of vertices attached to α , and the point of contact of these vertices with α are called the *ports* of α . There is a distinguished port, called the *principal port* which is the point of contact where communication can take place.
- A set of rewrite rules which are activated when two symbols are connected on their principal port. The notation $\alpha \bowtie \beta$ will be used to mean that two nodes are connected on their principal ports.

◇

The basic idea of an interaction rule is then given by the following diagram which shows $\alpha \bowtie \beta$.



Where $\mathcal{N}$ is a new net, which may contain occurrences of the original nodes in the interaction. Note that the interface is preserved; there are equal numbers of ports

before and after the interaction which are exactly the same free ports as the original net.

As an almost immediate consequence of this kind of rewriting we have the following features:

- **Confluence.** The restriction of communication only on the principal port of an agent, and the constraint that each node has only one such port suffices to give the strongest notion of confluence.
- **Local Implementation.** The diagram above shows that the local interface is preserved during an interaction — two agents interacting do so in their own “space” and do not affect any other part of the network.
- **Asynchrony.** As a direct consequence of the above point, we have the possibility of a truly parallel implementation — no order on the interactions is required. Any two agents ready to communicate can do so in any order. Reduction can take part anywhere in the net, and in particular, for the λ -calculus, we can have reductions within substitutions, for example, before they are fully propagated.

Interaction nets can be regarded as a generalisation of proof nets for Linear Logic, and indeed, this is their origin. Without being too precise, the relationship is given by setting the set of symbols Σ to be the *logical symbols*; the principal port of each symbol is the *conclusion* and the auxiliary ports are the premises of the rule; and the rewrite rules are specified by the rules for *cut-elimination* for Linear Logic; we will make this more precise below. In many senses Interaction Nets provide a programming paradigm for Linear Logic. By considering the translations given in Chapter 3 we immediately have the possibility of an interaction net implementation for the λ -calculus. Additionally, we can “fine tune” the interaction nets to yield a very simple system of only five agents and 11 rewrite rules.

We remark finally that Bawden [Baw86] has also considered a system of *connection graphs* in the spirit of interaction nets. In particular he gave the first coding of a functional language (Scheme) into these networks.

Before presenting our interaction net for the λ -calculus, we outline our methodology.

1. First we briefly outline an interaction net for the fragment of Linear Logic that we are using. This is the “synchronous boundary” Interaction Net presentation developed by Abramsky [personal communication].
2. For the chosen translation that we use for encoding the λ -calculus into Linear Logic proof structures, we observe that there are certain nodes that always

appear together. We hence define a simpler system by introducing “hybrid” agents which in fact turn out to be nothing more than just the lambda and application nodes that one would find in standard graph reduction, (see for example [Pey87]). Indeed we will see that the implementation that arises is very closely related to Wadsworth’s algorithm. We will return to this point at the end of this chapter.

5.2.1 Nets for Linear Logic

In this section we give an interaction net implementation of cut-elimination for Multiplicative Exponential Linear Logic. The implementation that we give differs from that of [Laf94] and [GAL92] in that we have a *lazy* (or outermost) implementation of the exponential box. We achieve this by a synchronisation mechanism which restricts interaction to only take place at the main door of the box. In addition, our presentation also is a finite one in the sense that we have a finite set of nodes that are used to code proofs. The advantage of a finite presentation is shadowed by the fact that we have to introduce additional nodes that are not needed in extant presentations. We come back to this point later when we consider the λ -calculus, where we will eliminate these extra nodes, but lose the finiteness property again. However we will be left with a very simple interaction net presentation of the λ -calculus.

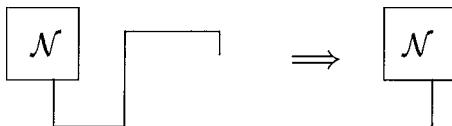
We begin by introducing the agents alongside the corresponding proof net rule of Linear Logic. There will be a new agent for each logical rule, each of which has edges typed as shown by the logical rules.

Identity Group

An Axiom and Cut are represented simply by a piece of “wire” in Proof Nets which will be the same representation in our nets—they connect agents. They are drawn respectively as:



A Cut of a net against an Axiom link is implicit by the fact that the concatenation of two pieces of wire is still a piece of wire:



Structural Group

Since we are working with networks with a 2D syntax there is no rule corresponding to the structural rule. In other words nets provide a more canonical notion of a proof.

Multiplicative Group

Tensor and Par are duals and this feature comes out very clearly in the net presentation. For each multiplicative connective we have an agent with two auxiliary ports, representing the hypotheses, and the principal port representing the conclusion.

Tensor The proof net for the $\otimes$ rule is given by the following configuration. We write proof nets following the original notation of Girard [Gir87]. If A and B are the conclusions of two nets, then we can combine them as follows:

$$\frac{A \quad B}{A \otimes B}$$

Corresponding to this rule we introduce the following $\otimes$ agent. We will omit types from the net, but they can be read off from the corresponding proof net rule above.



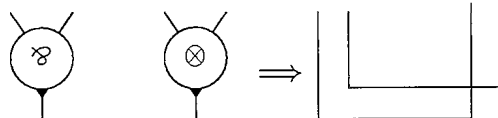
Par The proof net rule for a $\wp$ is identical to the above Tensor rule:

$$\frac{A \quad B}{A \wp B}$$

And the corresponding agent, $\wp$, is introduced is as follows:



The cut-elimination dynamics for the multiplicative fragment is given by the following $\wp \bowtie \otimes$ interaction:



We remark that we could collapse the presentation of the multiplicative fragment to just one agent, if we assume that the net is produced from a correct proof. In other words the interaction net is a representation of the structure of the proof, not the proof itself. This methodology is indeed used in the algorithm of [GAL92] and reduces the number of agents and reduction rules in their presentation. We prefer to keep these separate since it makes the nets easier to work with. The alternative presentation is certainly a useful coding trick for implementation.

Exponential Group

The exponentials are a little more complicated. The rules for Dereliction, Contraction and Weakening are straightforward, but the rule for Promotion is made complicated by the fact that we want to code a box in a local way. As with the multiplicative fragment, we generate a node for each proof net construction, where the premises are the auxiliary doors, and the conclusion is the principal door.

Dereliction This rule has one premise and one conclusion:

$$\frac{A}{?A}$$

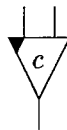
Hence we generate a new node d corresponding to dereliction as follows:



Contraction The rule for the contraction is similar to the binary multiplicative nodes:

$$\frac{?A \quad ?A}{?A}$$

Yielding a binary node for the contraction c , which we draw as a fan-in node (cf. Chapter 2) and distinguish the premises by marking one side of the fan node as shown:



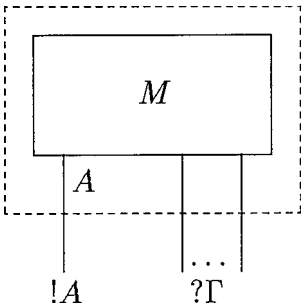
Weakening From no premises, we can conclude $?A$:

$$\overline{?A}$$

This suggests a unary node w which is drawn as follows:

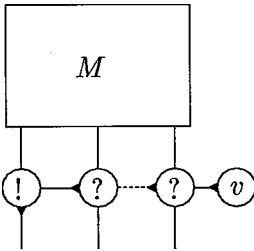


Promotion Our final compilation rule deserves a special treatment. The proof net representation of $!A$ is given by the following, where M is the proof net for A :



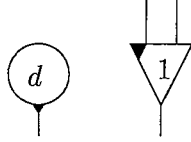
For an interaction net representation of this we first need to introduce a $!$ node which corresponds to the principal port of the box. For each auxiliary port, we add a $?$ node which represents the extent of the box. A list of these nodes is terminated by a *void* node v . The whole philosophy behind this implementation is given by the fact that we *connect* the auxiliary nodes to the main door of the box. All of our dynamics will make use of this connecting edge in a way that will become clear. This in fact will give us a *local* implementation of the global feature of proof nets.

Hence our translation of a proof $!M$ is given as the encoding of a term M with the box structure as explained, viz:



Dynamics

As we mentioned before, the cost of the finite presentation is offset by the introduction of new nodes during the cut-elimination process. In addition to the $?$ and v nodes already mentioned there are the following two nodes:



These are respectively a unary dereliction node d and a new fan node 1 , which we will call a *duplicator* node. These nodes have no logical counterpart, and we must therefore explain their purpose.

- The unary dereliction node is generated when a dereliction is cut against a box. This auxiliary node is used to interact with the side doors $?$ and remove them, thus completing the dereliction of a box.
- A duplicator node copies everything on its principal port.

They are generated when a contraction node is cut against a box: we push duplicator nodes in all sides of a box to copy its contents. By the interaction net principle of only communicating on a principal port, we observe that we only copy parts of the proof that are in normal form. So the idea is that duplicator nodes squeeze out the redexes in a term while they perform the copying.

The well balanced nature of the way we use duplicators means that whenever two duplicator nodes meet, they originate from the same fan node and hence should cancel each other out. Therefore we do not need indexing on these nodes, nor do we need additional nodes to change the level of depth that these nodes are operating at.

We hope that the reader will gain more insight into these comments after examining the dynamics of the system.

The interaction rules for the exponentials (and their interaction with the multiplicatives) are shown in Figures 5.1, 5.2 and 5.3. We overload notation a little and write m for a multiplicative node ($\wp, \otimes$), since the rules for a duplicator meeting these nodes are identical. Figures 5.4 and 5.5 shows the rules for the weakening node — our garbage collection rules. We again overload the notation and write i for both a contraction and a duplicator node which have identical behaviour on weakening nodes.

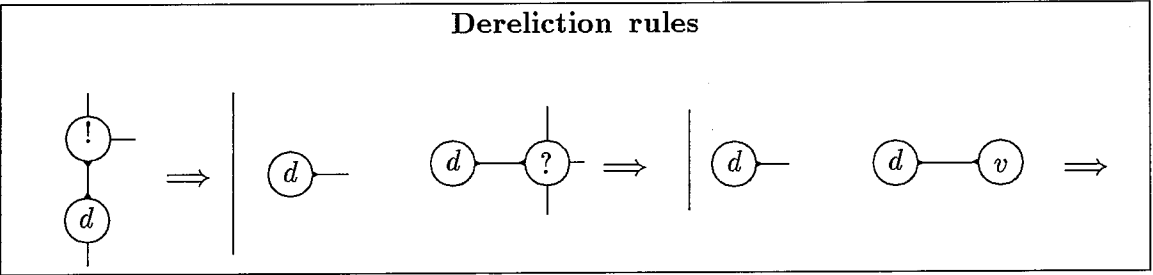


Figure 5.1: Dynamics of Cut elimination for Linear Logic

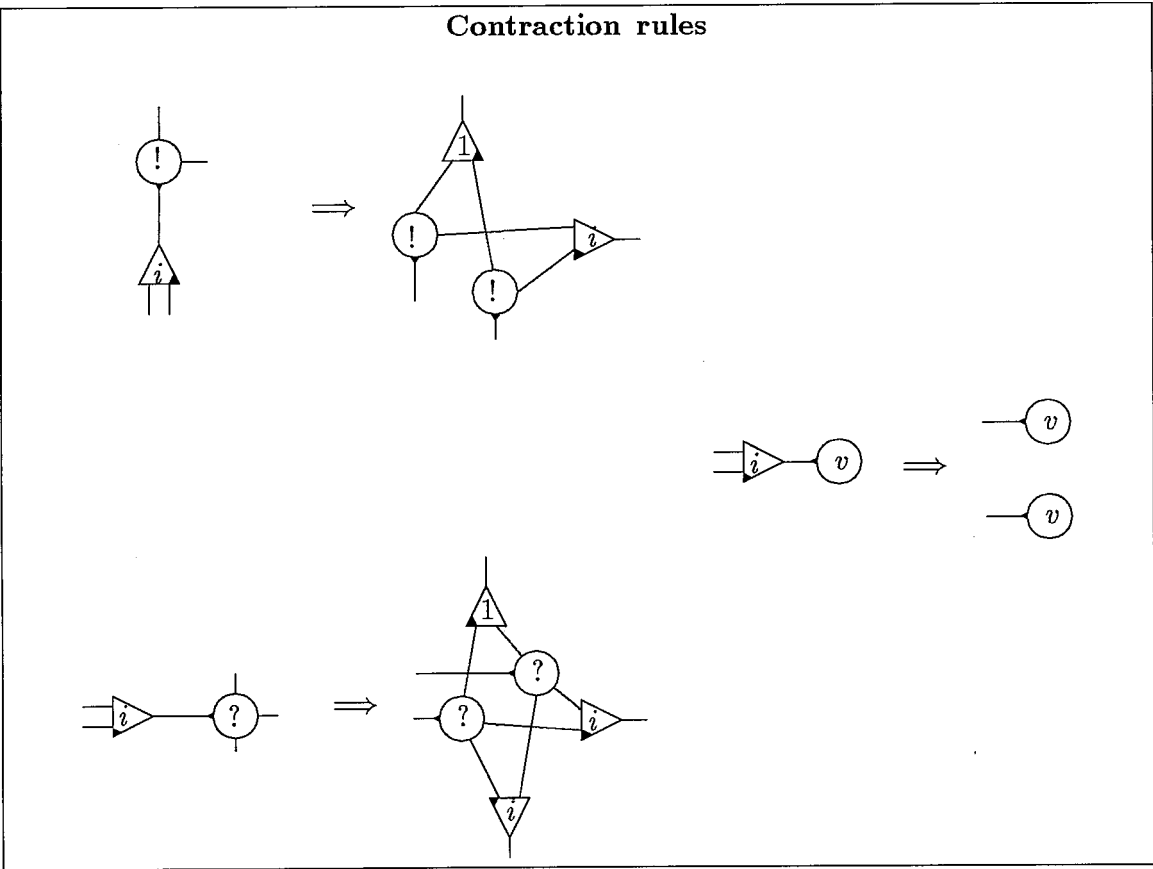


Figure 5.2: Dynamics of Cut elimination for Linear Logic, continued

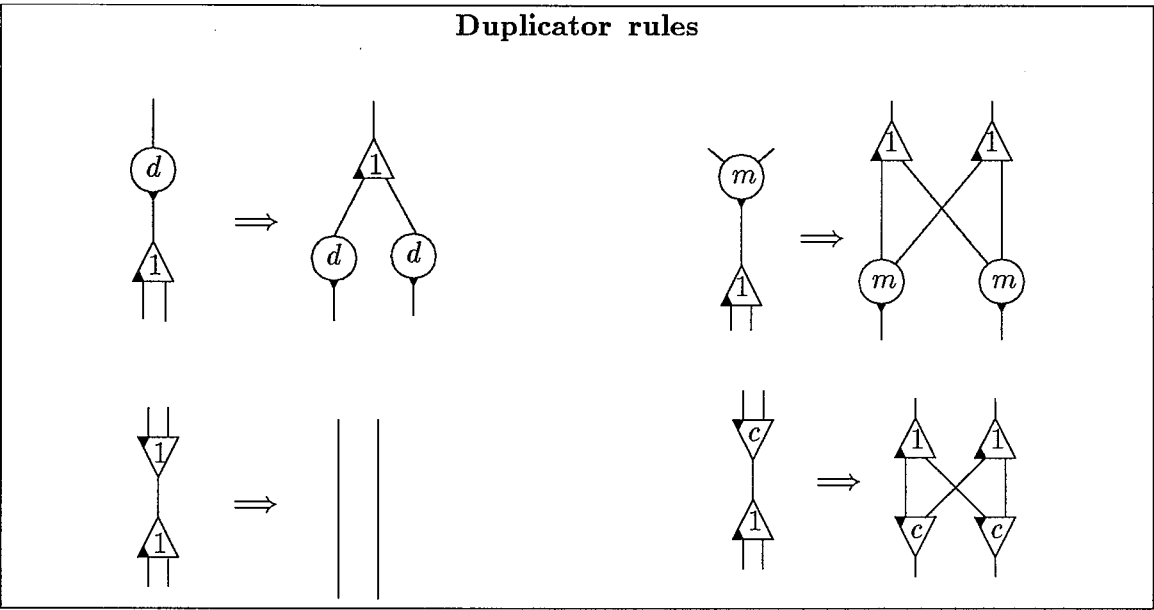


Figure 5.3: Dynamics of Cut elimination for Linear Logic, continued

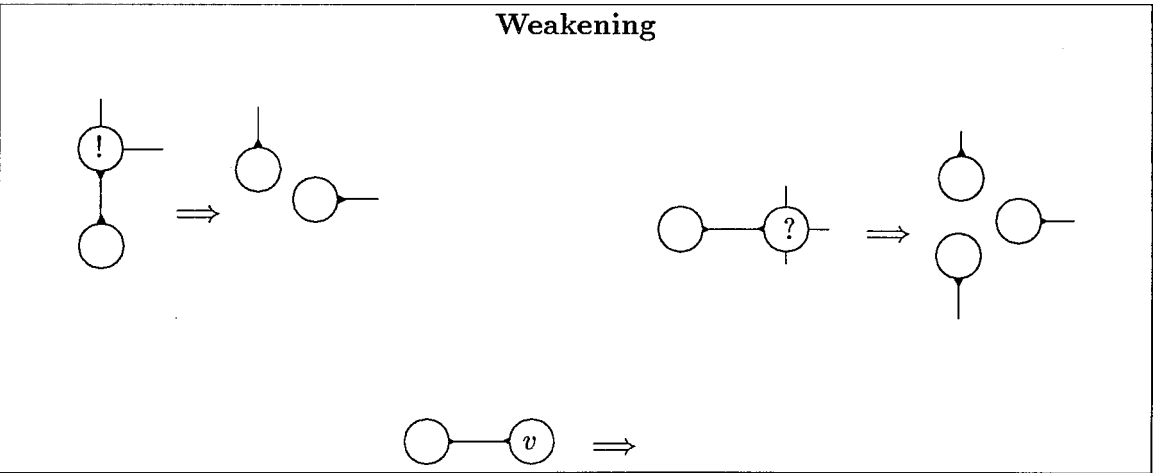


Figure 5.4: Weakening rules for Linear Logic

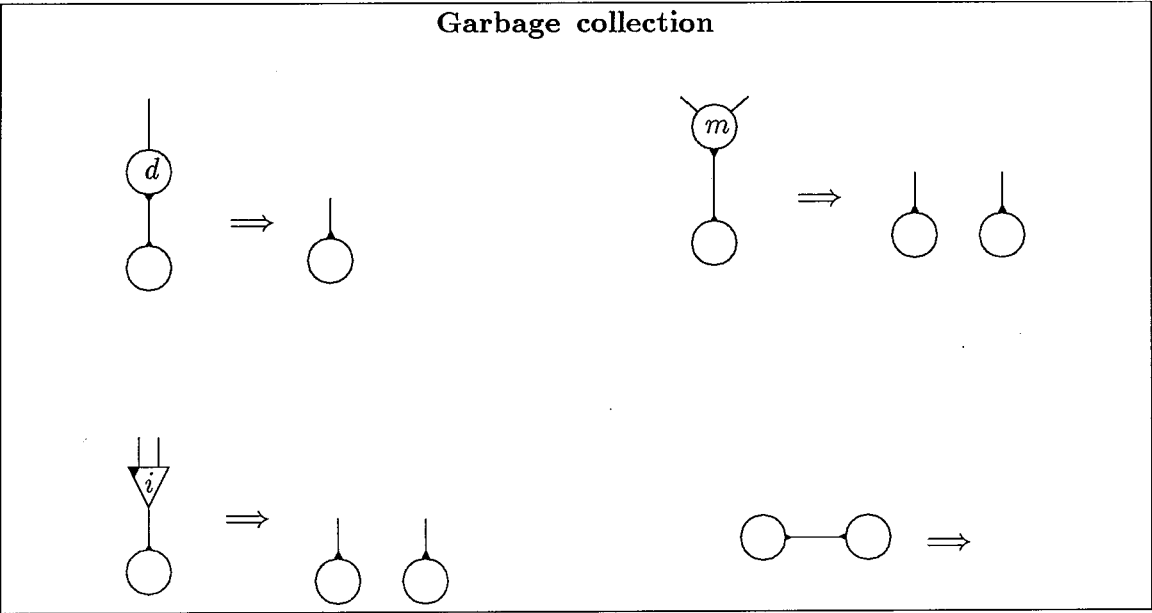


Figure 5.5: Weakening rules for Linear Logic, continued

As we have already stated, this implementation gives a *local* implementation of the *global* rules on boxes. It is because of this that we fail to get a notion of optimal reduction for these nets. We will come back to this point when we deal with the λ -calculus.

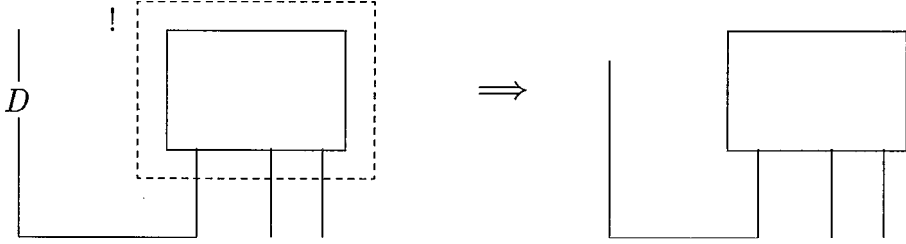
We also remark that we abused the use of the weakening node for *garbage collecting*. In other words, the type of this node given by the proof net rule states that it should only be cut against formulae of $!$ type and we have shown rules for it cut against non-exponentiated formulae. However, we reassure the reader that there is no harm in this, since the introduction of a new node would have exactly the same behaviour as a weakening node.

The right way to understand the dynamics of this system is to think that the local rules implement cut elimination in Linear Logic. We will prove this shortly, but first a description of the dynamics is required to give the reader some insight to what is going on.

The proof net reductions can be explained in terms of a sequence of interactions given by the dynamics. We explain each of the proof net cuts in turn. For each cut-elimination rule in Linear Logic proof nets we show that the interaction net representation simulates this cut by a sequence of interactions.

Derelection-! The required action of a derelection cut against a box is to remove the box, leaving the contents correctly connected to the rest of the net. The

proof net cut is given by:



We simulate this in our interaction nets by *three* kinds of interaction given in Figure 5.1.

$d \bowtie !$ The process begins here by the interaction of the dereliction and $!$ nodes.

This interaction removes the $!$ node and connects the contents of the box to the outside world. The auxiliary (unary) dereliction node is then introduced, connected to the list of $?$ side-door nodes.

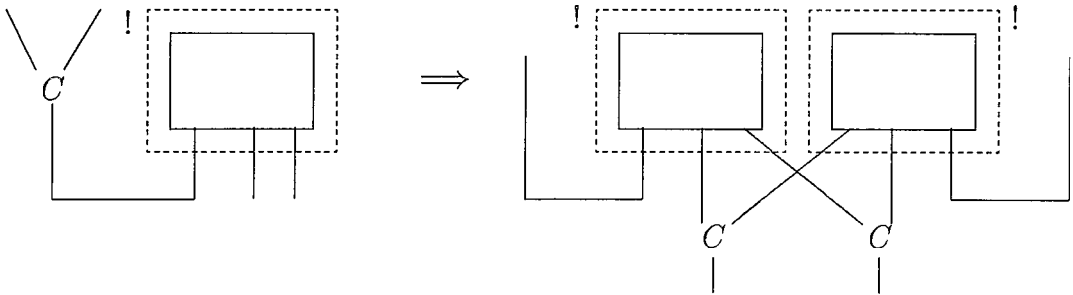
$d \bowtie ?$ The unary dereliction node is propagated along the list of side door nodes to complete the opening of the box. Hence the interaction connects one of the auxiliary edges from inside the box to the outside world, and the unary dereliction node is propagated onto the next side door.

$d \bowtie v$ At the end of a list of side doors, we have a void node. The dereliction is completed when the unary dereliction node reaches this void node. We remark that these nodes annihilate each other so that there are no nodes on the right hand side of the interaction.

Intuitively, the above three interactions can be thought of as lighting a *fuse* wire on interaction $d \bowtie !$ which erases all of the side doors (rules $d \bowtie ?$ and $d \bowtie v$).

Hence the cut is simulated *exactly* in the interaction net presentation.

Contraction-! The required action of a contraction cut against a box is to copy it, and copy all the auxiliary doors. The proof net cut is given by:



We simulate this in two parts — the copying of the box structure, and the copying of the contents of the box. These are shown in Figures 5.2 and 5.3.

Copying the box structure is the simplest to explain since we follow the same ideas as the sequence of interactions for the dereliction that we described above. Since we overloaded the notation for the interaction rules, we explain the scenario by instantiating c for i for the rules in Figure 5.2.

$c \bowtie !$ The process of copying the box structure begins here. This interaction copies the $!$ node, then, to maintain connectivity, we introduce a duplicator node 1 to connect to the contents of the box, and we send the contraction node along the list of side door nodes.

$c \bowtie ?$ The contraction node is propagated along the list of side door nodes, copying each one. To maintain connectivity, we introduce a duplicator node 1 to connect to the contents of the box, send a contraction node along the remaining list of side doors, and send a contraction node along the free edges of the box to copy these.

$c \bowtie v$ The process of copying the box structure is completed by the interaction of the copying and void nodes. This interaction simply copies the void node.

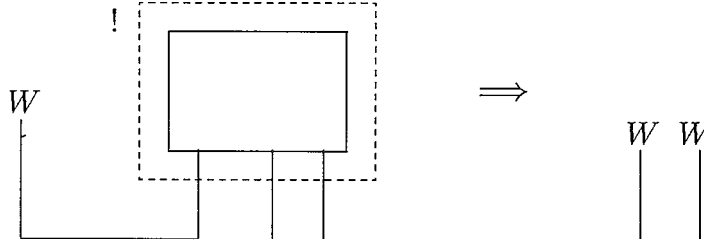
To complete the contraction we must copy the contents of the box, and this is the purpose of the duplicator nodes. We have a rule for a duplicator interaction against every node in the system, and they are given in Figure 5.3. The idea of duplication is simple in that we just copy the node involved in the interaction and propagate the duplicator nodes to complete the copying process. Propagation is completed by the rule $1 \bowtie 1$ when two duplicator nodes meet, in which case they annihilate each other.

The case of a duplicator node interacting with a box structure proceeds exactly as prescribed above, instantiating 1 for i in the rules given in Figure 5.2. Since we always introduce duplicator nodes within a box in a well balanced fashion we are guaranteed that when two meet they should annihilate each other—they are working at the same level of the environment.

A closer inspection of the rules show that duplication can only take place at the principal port of the nodes—it is an interaction net of course. The consequence of this is that the duplication process cannot copy any cuts within a box, except an $!-!$ cut. Hence the propagation of the duplicator nodes will only complete if the net inside a box is cut free; the process of duplication will cause the net to be reduced to normal form.

This final point leads to a slight mismatch between cut elimination in Linear Logic and the interaction net implementation. However, this mismatch is exactly the interesting point of this implementation—we have an implementation that will reduce a proof net to a cut free proof by performing fewer cut elimination steps by *sharing* the computation required to reduce the contents of the box. Hence all we are doing is enforcing a strategy on cut elimination.

Weakening-! The required action of a weakening cut against a box is to delete the box, and delete all the auxiliary doors. The proof net cut is given by:



We simulate this again in two parts—deleting the box structure, and then deleting the contents of the box. The interaction net rules are shown in Figure 5.4.

$w \bowtie !$ The process of deleting the box structure begins here. This interaction simply deletes the $!$ node, and sends a weakening node along the list of side doors and within the box.

$w \bowtie ?$ The weakening node is propagated along the list of side doors deleting each one and sending a weakening node both inside the box and along the edges of the side doors.

$w \bowtie v$ The process of deleting the box structure is completed by the interaction of the weakening node and the void node. The interaction simply deletes the void node; note that there is no right-hand-side to this rule—no new nodes are generated.

Similarly to the duplicator nodes, we introduce these nodes in a well balanced fashion that guarantees that propagation will proceed until the entire box is erased. Analogously to the way duplication of a box forces evaluation first, we have the same phenomenon here. Hence the rules force evaluation to normal form before the contents of the box can be erased.

Finally, we remark that the commutative conversion, the $!-!$ cut, is *not* modelled by this interaction system. This is the essence of the reason that we fail to achieve optimal reduction. It is the only reduction that we fail to implement.

5.3 Nets for the λ -calculus

Having set up the machinery for the interaction net implementation of Linear Logic we can directly get an implementation of the λ -calculus using any of the translations that we gave in Chapter 3. However, rather than directly following the encodings we will fine tune the implementation for our specific needs. This will lead to a much simpler system.

As discussed in Section 3.5 we will base our implementation on the $D =!(D \multimap D)$ translation since this translation allows us to use the least number of nodes in our interaction net implementation. We will economise even further by observing that certain nodes always appear together; refer to Table 3.2.

- Abstractions are translated by $T[\lambda x.M] =!(\wp(T[M]))$. This indicates that we use a par node and an exponential box as described above. However, we observe that none of the re-writing rules splits these nodes, i.e. they always stay together during reduction. Additionally, the auxiliary nodes for the side door (the “?” nodes from before) need not be separated from the main door of the box. Hence we introduce a hybrid node which combines all these, which we will call a *lambda* node. Note then that we no longer have a finite system — we require a set of these nodes indexed by the number of free variables in the box.
- Applications are translated by $T[MN] = T[M] \cdot_x D(\otimes(I, T[N]))$. This indicates that we use a tensor node and a dereliction node as described above. Again, however, we observe that these two nodes are inseparable and introduce a new node which combines these. This will be called an *application* node.

We can now present a much simpler algorithm with only five nodes and 11 interaction rules. Using the above observations we feel also that the interaction nets are more understandable and correspond more intuitively with standard graph presentations of the λ -calculus.

The basic agents for the interaction net are given in Figure 5.6, showing the principal ports. These nodes are exactly what we have been using so far, but using the hybrid nodes mentioned above. As before, we overload notation a little and write i for a either a contraction node (c) or a duplicator node (1). Hence the five nodes are *contraction*, *duplication*, *weakening*, *application* and *abstraction*.

5.3.1 Translation

We now give the translation $\mathcal{T}$ of the Linear $\lambda\sigma$ -calculus into Interaction Nets. More specifically, we are compiling these terms into Linear Logic proof structures using

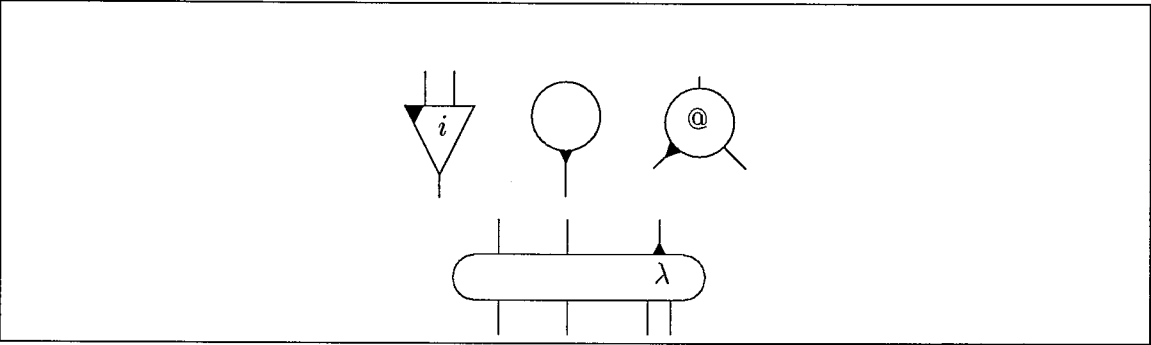


Figure 5.6: Interaction Net Agents for the λ -calculus.

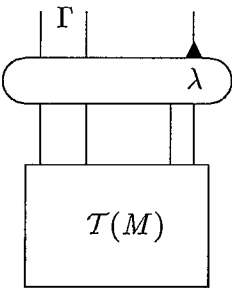
the $D =!(D \multimap D)$ translation; we will mention also the $D =!D \multimap D$ and $D =!D \multimap !D$ translations at the end of this chapter.

- 1. A variable x is translated into a connecting edge (a piece of “wire”). No new nodes are required under this translation. We draw this simply as:

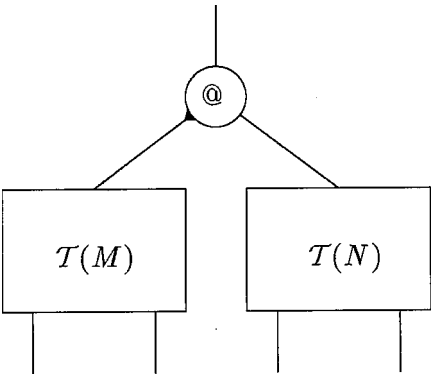


We will often write the name of the edge with the corresponding variable in the λ -calculus to aid readability, but this is not at all necessary.

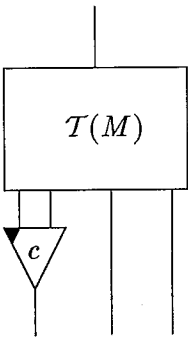
- 2. An abstraction $\lambda x.M$ is translated into a “boxed” structure as drawn below. The wires from left to right are $\Gamma = fv(M) \setminus \{x\}$, the root of the term M , and finally the binding variable x is the *rightmost* edge.



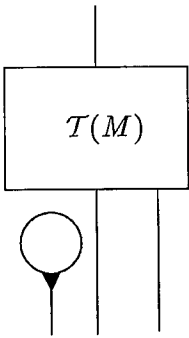
3. An application MN is translated directly into an application node.



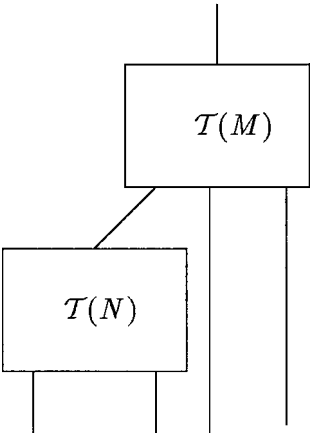
4. The contraction of u and v to z ($[z = \langle u, v \rangle]M$) is compiled into the following structure using the contraction node.



5. The weakening of the variable x in M ($[x = _]M$) is compiled into the following structure using the weakening node.



- 6. A substitution $M[N/x]$ is coded in the obvious way, where here we assume that the free variable x is the leftmost free variable in M .

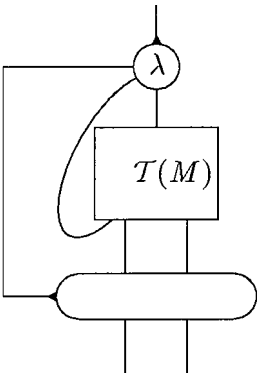


A few notes on the translation:

- 1. A variable is translated into a piece of “wire” connecting agents. More formally, this piece of wire is given a name corresponding to the variable, and the ends of the edges should be seen as having plugs at either end with the following types:

$$A^\perp \text{ --- } x \text{ --- } A$$

- 2. The author has found it fruitful to draw abstractions as follows:



which resembles a little more the standard graph representation of the λ -calculus. One should note, however, that the λ node and the side doors are all one node. We could have presented the algorithm in many different ways by separating these nodes, however this leads to requirement of additional nodes and rewrite

rules. It was our philosophy to keep things as simple as possible in this algorithm.

To give the reader some insight into this translation, we show the translated versions of the combinators I , K and S , (see Example 2.2.17), in Figure 5.7.

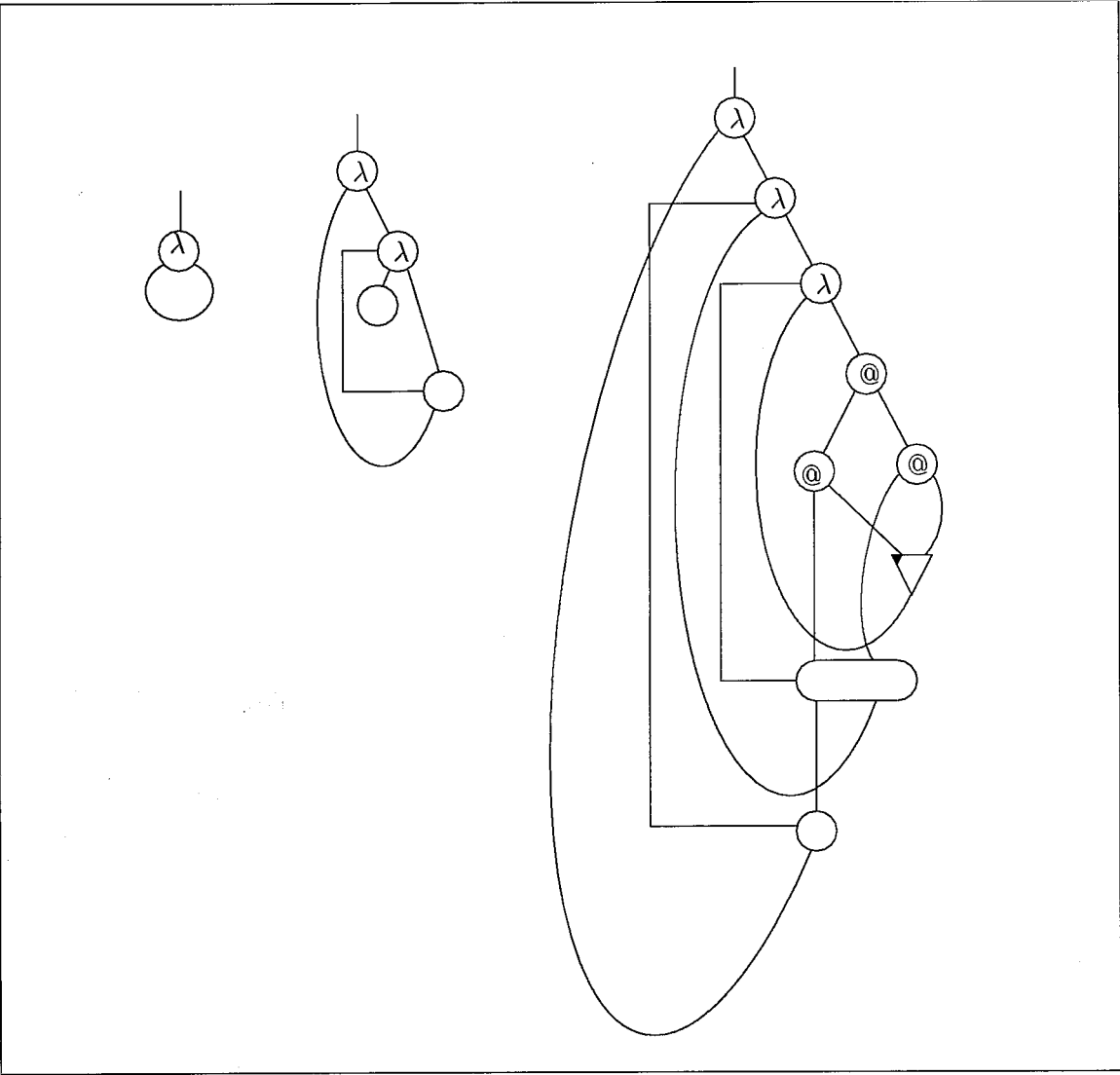


Figure 5.7: Examples of graphs generated: I , K and S

5.3.2 Dynamics

In this section we give the valid interaction net rules for our system. There are 11 rules in all, of which five are for garbage collection. We simplify our presentation before by observing that the rules for contraction and duplicator are the same for

two of the rewrite rules; we represent this by writing i to represent either contraction (c) or duplicator (1). The rules are given in Figure 5.8, and Figure 5.9 shows the garbage collection rules.

We refrain from giving a detailed explanation for the interaction net rewrite rules for the λ -evaluator which are just compositions of the rules for implementing Linear Logic proof structures that we presented in the previous section. The compositions are as follows:

- The $\lambda \bowtie @$ interaction is a combination of the $d \bowtie !$, $d \bowtie ?$, $d \bowtie v$ and the $\otimes \bowtie \wp$ interactions.
- The $\lambda \bowtie c$ interaction is a combination of the $c \bowtie !$, $1 \bowtie \wp$, $c \bowtie ?$ and $c \bowtie v$ interactions; and similarly for $\lambda \bowtie 1$.
- The $1 \bowtie @$ interaction is a combination of the $1 \bowtie D$ and $1 \bowtie \otimes$ interactions.

The interactions $1 \bowtie 1$ and $1 \bowtie c$ are exactly as before.

We remark that the set of interactions is complete in the sense that all other possible interactions (for example the interaction of an application node against an application node) do not have well typed edges, and do not make sense under cut elimination. Hence a net arising from the translation given will never be in a state such that two principal ports are ready to communicate but there is no rule for the interaction. The case that a net is not in normal form, but there are no reduction rules for the remaining interactions is called a *blocked* net.

We point out some of the most significant differences between this algorithm and the extant interaction net algorithms for the λ -calculus:

- We do not perform commutative conversions—we restrict the rule $!(P) \cdot_x !(Q) \rightarrow !(P \cdot_x !(Q))$. Back in the λ -calculus under the $D = !(D \multimap D)$ translation this means that we do not push substitutions into an abstraction; under the $D = !D \multimap D$ this means that we do not push substitutions into *arguments* — just into head variables of a term which means that we reduce to principal head normal form.

This notion of weak substitution is reminiscent of the restricted normal forms investigated in the context of the $\lambda\sigma$ -calculus [ACCL91]. We propose to call this *restricted substitution normal form (RSNF)*. Our implementation is a very simple, natural, coding of Linear Logic proof structures into Interaction nets,

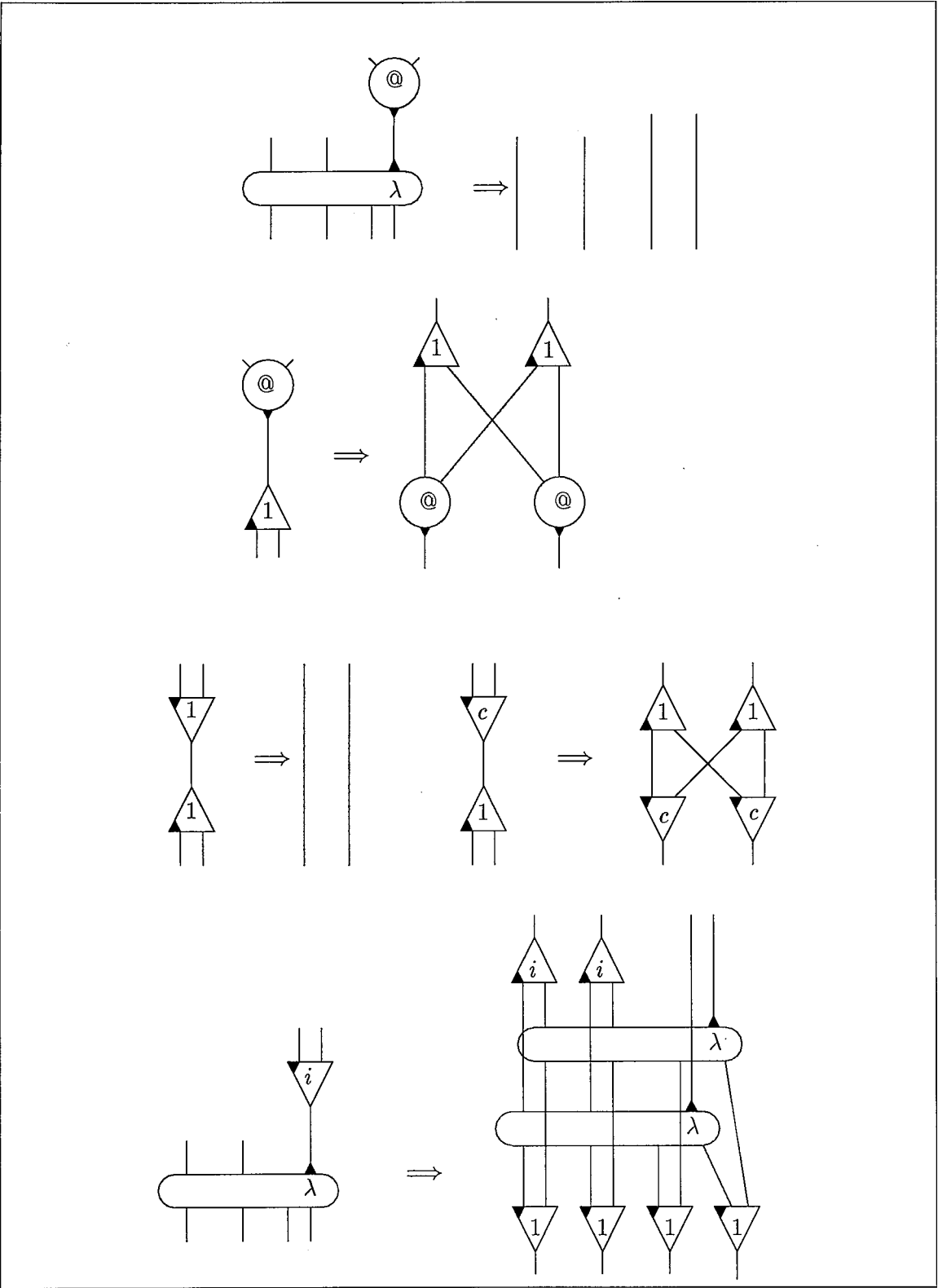


Figure 5.8: Dynamics

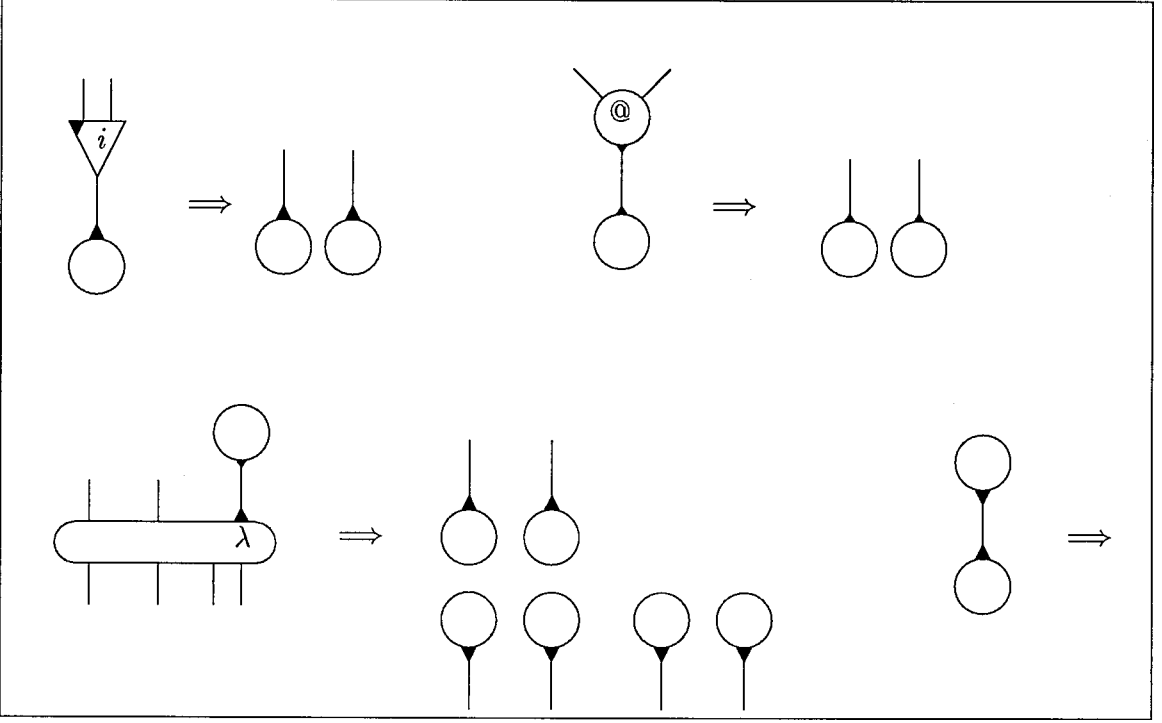


Figure 5.9: Garbage Collection

and we feel therefore that we should not be disturbed by this notion of normal form.

- As a consequence of the above, we do not need to have infinite indexing on the duplicator nodes. Indexing is required to code the level of the environment which the nodes are operating on. Since we have the side door connecting edge we know that everything is balanced. This makes the dynamics and the semantics simpler; the system is in fact just performing cut elimination.
- On a negative note, we do *not* get optimal reduction — we explain why when we analyse the algorithm.

5.4 Properties of reduction

Here we will look at some general properties of our translations and reductions to justify the correctness of the algorithm.

Proposition 5.4.1 (Subject reduction)

All interaction net rules preserve types on the edges.

◇

PROOF: A straightforward analysis of the interaction net rules indicates that the types of all the edges are preserved. $\square$

This notion of *semantic soundness* or *nothing will go wrong* formalises our previous comment that our nets will never *block*, hence interactions will always be possible until the net has reached a normal form (if one exists).

This property is due to the fact that we are doing nothing more than implementing the cut elimination rules for Linear Logic. The only point of departure is the local copying of the contents of a box in the contraction-! cut for which we used the duplicator nodes. Therefore, to prove that this algorithm is *correct* all we are required to do is show that the duplication process is well behaved. In particular, the following are the required properties:

- First, we need to show that the duplicator nodes stay well balanced during reduction.
- Secondly, we need to show that the duplication process terminates so that the contraction-! cut is correctly implemented

We want to show that each of the rewrite rules on our interaction net are sound with respect to some “denotational” semantics.

Since we are using Linear Logic as a foundation of our algorithm, we can appeal to Girard’s Geometry of Interaction interpretation of proofs [Gir89a]. This dynamic semantics of cut elimination provides a very elegant and simple analysis of our nets. More specifically, we will use the notion of a *path* in a λ -term, and use the dynamic algebra Λ^* following the notation of a regular path of Danos and Regnier [DR93] which we recalled in Section 4.5.

The outline of the mechanics of the correctness proof is as follows:

- The entire net can be given a denotation in the Geometry of Interaction interpretation of Linear Logic as a set of execution paths.
- Each local rewrite rule is shown to preserve all paths — all paths survive the action of an interaction, and no new paths are generated.

By the compositional nature of this interpretation, this local correctness gives correctness at a global level.

Before we give the proof, we have to extend the interpretation to include the duplicator nodes. A duplicator node is a contraction node, but working at a greater level of environment. We definitionally extend the algebra Λ^* with duplicator

coefficients r_k and s_k such that they satisfy $!^k(r_k) = r$ (and similarly for s) in any context. More generally, we have $!(r_{k+1}) = r_k$.

The main result can now be stated as follows.

Lemma 5.4.2 (Persistency)

For each interaction rewrite rule $N \rightarrow N'$ in Figure 5.1 we have that if ϕ is a path in N , then there is a corresponding path ϕ' in N' such that if $\phi \neq 0$ then $\phi' \neq 0$ and no new paths are generated. Hence all paths survive the interaction; they are persistent. $\diamond$

PROOF: We show this by cases on the rewrite rules.

1. $\lambda \bowtie @$: there are 3 kinds of path to consider:

(a) Type I. Before the interaction we have the following path, which we reduce to its normal form:

$$\begin{aligned} p^*d^*!(p\phi q^*)dq &= p^*d^*dp\phi q^*q \\ &= p^*p\phi \\ &= \phi \end{aligned}$$

After the interaction, we have simply ϕ . Hence, the path survives the action of the cut, and the paths are equivalent.

(b) Type II. Again, we show the path before the interaction, and reduce it to normal form:

$$\begin{aligned} t!(\phi q^*)dq &= td\phi q^*q \\ &= td\phi \end{aligned}$$

After the interaction, we are left with ϕ , and it is trivial that $\Lambda^* \vdash td\phi \neq 0 \implies \Lambda^* \vdash \phi \neq 0$.

(c) Type III. Again, we show the path before the interaction, which, this time, is already in normal form.

$$t!(\phi)t^*$$

After the interaction, we are left with ϕ , and again it is trivial to verify that the path is non 0.

2. $c \bowtie \lambda$: Again, there are three kinds of path to consider for each side of the contraction node. We show the cases for s :

(a) Type I. Before the interaction we have a path $s^*!(\phi)s$ which reduces to $!(\phi)$. After the interaction we have the path $!(s_1^*\phi s_1) = s^*!(\phi)s = !(\phi)$ as required.

(b) Type II. Before the interaction there is the path $t!(\phi)s = ts!(\phi)$, for some path ϕ within the box. After the interaction there is the path

$st!(s_1^*\phi s_1) = sts^*(\phi)s = st!(\phi)$. It is straightforward to verify that this path survives the interaction.

- (c) Type III. Before the interaction there is the path $t!(\phi)t^*$, for some path ϕ within the box. After the interaction there is the path $st!(s_1^*\phi s_1)t^*s^* = sts^*(\phi)st^*s^* = st!(\phi)t^*s^*$. It is straightforward to verify that this path survives the interaction.

All the other possible paths follow the same arguments.

3. $1 \bowtie \lambda$: The proof proceeds in exactly the same way as above.
4. $1 \bowtie @$: There are four paths to consider; we just show one path for the case of s_k^*dp which rewrites to dps_k^* as required. The other three cases are similar.
5. $1 \bowtie 1$: Before the interaction, we have two paths: $r_k^*r_k$ and $s_k^*s_k$. Both of these paths reduce to 1, which are the weights of the paths after the reduction.
6. $c \bowtie 1$: There are four paths to consider; we just show the case for $r_k s^*$ which rewrites to s^*r_k which is the weight of the path after the interaction as required.

□

Proposition 5.4.3 (Well balanced duplicator nodes)

Let $\mathcal{T}(M) = N$ and then $N \rightarrow^* N'$ be a sequence of interaction net rewrites, then all duplicator nodes are balanced (“well bracketed”). ◇

PROOF: The proof proceeds by an induction on the number of interaction net rules applied to a net.

First, we observe that an initial net derived from the translation given does not introduce any duplicator nodes, hence our claim holds vacuously for zero applications of the rewrite rules.

We now show, under the hypothesis that a net N has well balanced duplicators after n interactions, that the proposition holds for any additional rewrite rule that can be applied. By Lemma 5.4.2 all paths are preserved under reduction; we show that all these paths preserve the bracketing condition: if we write (as we enter a duplicator node from the top and) from the bottom, then all reductions preserve the parity. There are six cases to show:

1. $\lambda \bowtie @$: This rule does not involve creation or elimination of any duplicator nodes, hence all paths remain well balanced.
2. $1 \bowtie @$: All four paths are preserved, and each path has one (both before and after the reduction, hence well balanced.

3. $c \bowtie 1$: Similar to the above case.
4. $1 \bowtie 1$: Both paths are preserved, and each path has $()$ before and no duplicator nodes after an interaction. Since the reduction eliminates a well balanced pair the resulting net is well balanced.
5. $1 \bowtie \lambda$: Again, there are three kinds of path to consider:
 - (a) Type I: Let ϕ be the well balanced path from the principal port back to the principal port before the interaction. After the interaction we have (ϕ) within the box, which is a well balanced path.
 - (b) Type II: Let ϕ be the well balanced path from the principal port to an auxiliary port. Before the interaction we have the path ϕ . After the interaction we have $) (\phi)$ so (ϕ) is a well balanced path within the box, and the entire path preserves the property.
 - (c) Type III: Let ϕ be the well balanced path from an auxiliary port to an auxiliary port. After the interaction we have we have the path (ϕ) within the box, which is well balanced.
6. $c \bowtie \lambda$: There are three kinds of path to consider:
 - (a) Type I: Let ϕ be the well balanced path from the principal port back to the principal port before the interaction. After the interaction we have (ϕ) within the box, which is a well balanced path.
 - (b) Type II: Similar arguments to above.
 - (c) Type III: Again, similar to above.

□

5.5 Extensions

Here we make the work a little more practical by showing how we can extend our implementation to handle features of real programming languages. We begin by extending the language with PCF-like constants. We will then briefly mention additional features, such as lists, so that we get an understanding of how to code larger languages.

The first restriction is that we are now working in a *typed* framework. The translations given in Chapter 3 are exactly the same where we replace D by a specific type. The translation is then restricted to simply typed terms.

PCF Constants

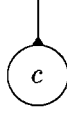
Recall the coding of PCF constants into Linear Logic proof structures that we gave in Chapter 3, in particular the “ $!(A \multimap B)$ ” translation given in Table 3.5.

It is possible to add the constants into this interaction net framework directly as given by the translation. However, we will continue our programme of trying to keep the resulting net as simple as possible and to this end we will introduce hybrid nodes in the same spirit as we introduced the λ and $@$ nodes.

Constants at base type

The constants $c \in \{n, \mathbf{tt}, \mathbf{ff}\}$ are all coded as promoted values. This indicates a node for each constant and then we have to build a box structure around these for our interaction net implementation. However, it is the case that the constant and the box always stay together so it seems reasonable to build a hybrid node of these two.

Hence for each constant $c : \tau$ we introduce a new node. Its only port is the principal port, which has type $!\mathbf{bool}$ or $!\mathbf{nat}$. We will slightly abuse notation and overload the nodes which will simplify our presentation.



Arithmetic functions

The constant functions **succ**, **pred** and **iszero** are coded by derelicting their arguments and promoting the result. Again, we observe that the function, dereliction and the promotion always stay together during reduction and thus introduce a hybrid node built from these components. Hence, for each constant f we have a binary node which we will draw as:

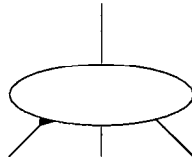


The intended meaning of the ports are: the result (at the top); and the principal port, where communication takes place with an integer (at the bottom). Types can be assigned to these ports depending on the function f ; for example **iszero** will get a typing $?\mathbf{nat}^\perp$ for the principal port and $!\mathbf{bool}$ for the result.

Note that the principal port of these hybrid nodes is the argument of the function—the principal port of the actual function rather than of the enclosing box. This is in fact a little trick that we can do with these hybrid nodes to overcome the fact the our interaction net implementation does not implement the commutative cut $!(P) \cdot !(Q) = !(P \cdot !(Q))$ which is required for the delta rules of PCF under this translation.

Conditional

The conditional is coded by derelicting the boolean argument and promoting the result. Therefore we define a hybrid node which is built from these components.

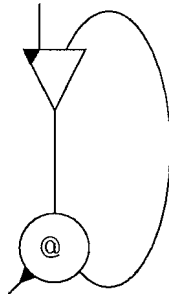


Clockwise from the top the ports are: result, false branch, true branch and finally the principal port: the boolean test. Hence a conditional node will only communicate with the result of the boolean test. Again, we can type the ports in the obvious way: result $!\sigma$, boolean test $?\mathbf{bool}^\perp$ and the true and false branches $?\sigma^\perp$.

Again, we have overcome the technical problem of the commutative conversion in this hybrid node by making the principal port the boolean test rather than the the result.

Recursion

We code recursion without the need to introduce any new nodes. The following cyclic structure explicitly “ties the knot”.



This corresponds exactly to the coding of recursion in graph reduction, see [Pey87]. We now get the coding of the PCF constant Ω as YI which is represented as the

following cyclic structure after one interaction:



Note that we are representing non-termination by deadlock! This means that our implementation will terminate more often, but the problem has now been pushed into the read-back phase of the implementation. We remark also that this coding of recursion is by no means fixed to PCF; it would work in the untyped setting too.

Aside 5.5.1 (For readers familiar with the cyclic λ -calculus)

Here we mention a nice application of these cyclic nets with respect to the Cyclic λ -calculus [AK93].

In the cyclic λ -calculus we can write terms as: $\alpha = \lambda x.M$, where α can occur in M . The simplest example being $\alpha = \alpha$, others being $\alpha = \lambda x.\alpha$ etc.

Graphically, these two examples are drawn as:



By making the *fan-in* explicit we find that we get exactly the same kind of cyclic structures that we are dealing with. In particular, the coding we gave above of YI is exactly the representation of $\alpha = \alpha$, and $\alpha = \lambda x.\alpha$ would be coded as YK .

In the cyclic λ -calculus there is a known counter-example of the Church-Rosser property which arises. Since we claim that we can represent this calculus in our nets there seems to be a mismatch — our reduction is based on interaction nets which possesses the strongest notion of confluence.

In fact the difference is simply expressed by the copying phase of a reduction—in our interaction nets implementation copying forces evaluation. We invite the reader to formulate the counter example in this framework to justify this claim.

This aside is intended to just show that there are no confluence problems with our cyclic structures. $\diamond$

Dynamics for PCF

We now show how these additional nodes react with each other. We assume terms are well typed which means that there is no possibility for an interaction of, for example, **succ** and **tt**.

There are four delta rules to this rewrite system. For the arithmetic functions we overload the notation and write f for **succ**, **pred** and **iszero**, and c for a constant. For the conditional, we show just the rule when the boolean argument is true. The other case is symmetric. The interactions are given in Figure 5.10.

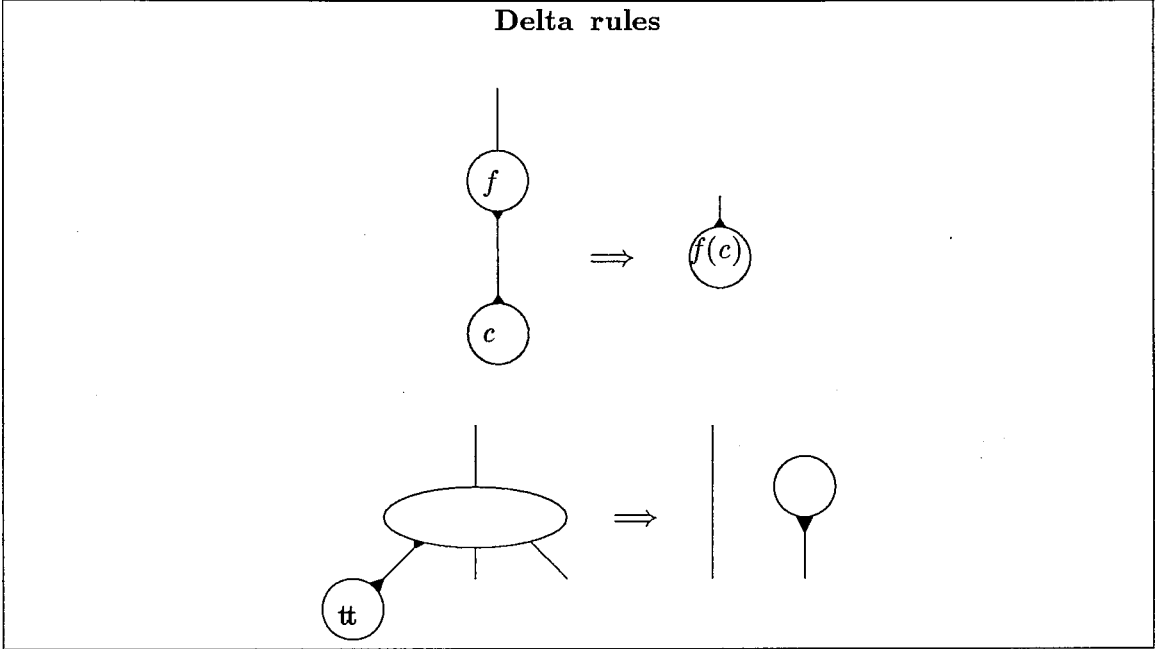


Figure 5.10: PCF Dynamics

The rules for **succ** $\bowtie n$, **pred** $\bowtie n$ and **iszero** $\bowtie n$ are straightforward in that they just apply the corresponding delta rule from PCF and introduce a new node for the constant.

The conditional deserves a careful treatment. Intuitively, this rule is straightforward in that we simply connect the result of the conditional to the appropriate branch, and then introduce a weakening node to garbage collect the unused branch. However, recall from Section 3.4 that the cut of a conditional against a boolean argument requires the application of the η rule which is indeed encoded into this rule. We remark that by introducing these hybrid nodes we are able to hide some of the details of the cut elimination process which would otherwise cause great difficulties in an interaction net framework.

We next need to show how these PCF nodes interact with the existing nodes. These are given in Figures 5.11 and 5.12.

The interaction rules for garbage collection are straightforward in that each rule simply deletes the node on interaction and weakening nodes are introduced on all the auxiliary ports of the deleted node.

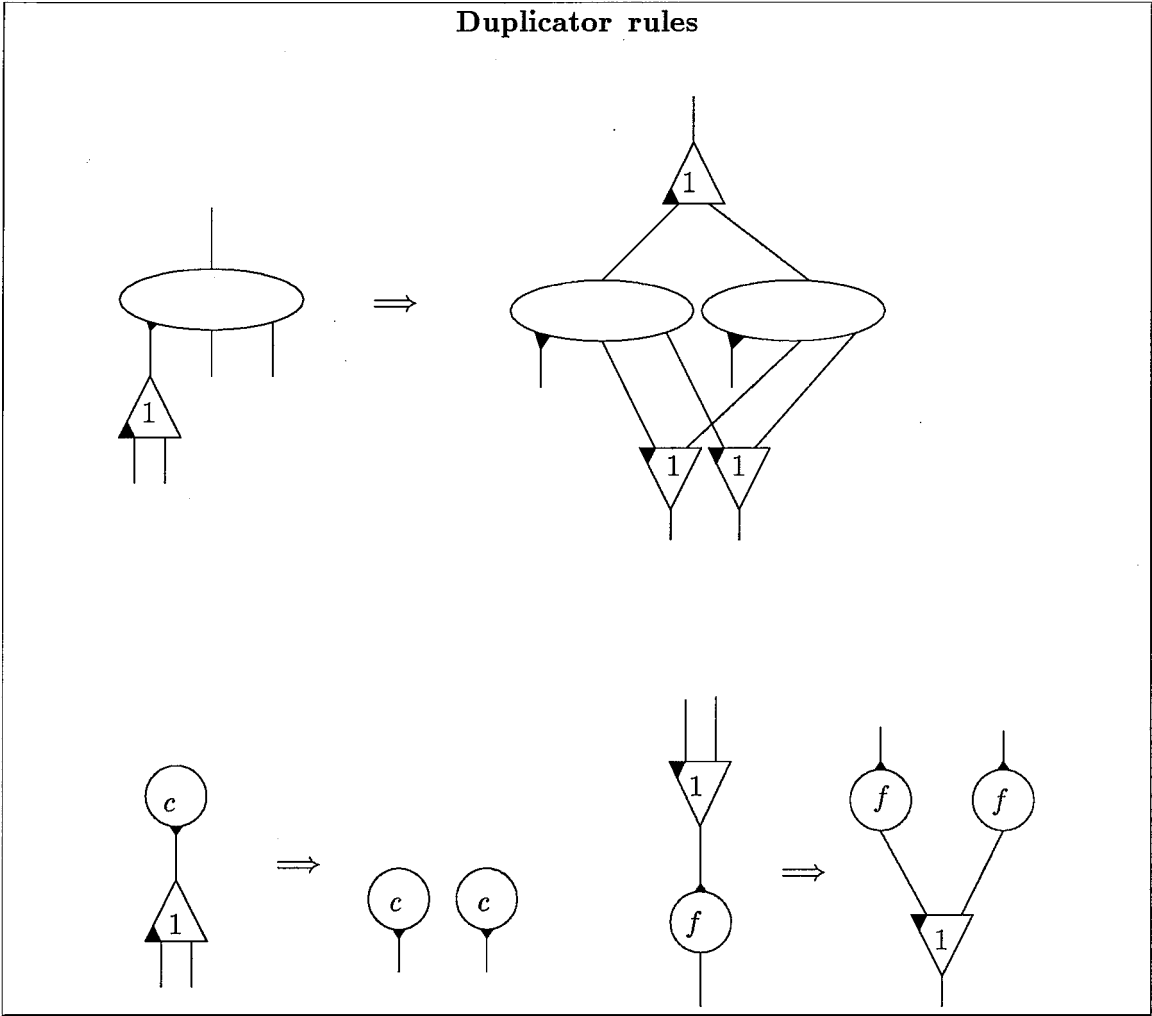


Figure 5.11: PCF Dynamics, continued

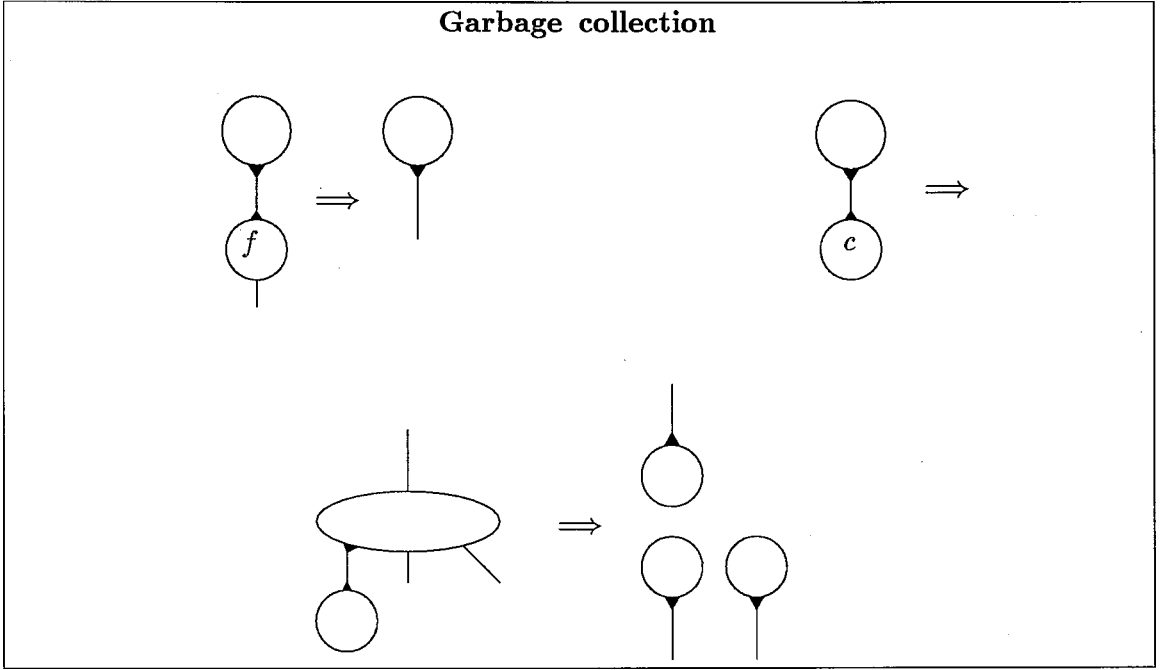


Figure 5.12: PCF Dynamics, continued

The duplication process of the PCF constants are again straightforward; they simply copy the node and propagate throughout the network.

Correctness for PCF extensions

Theorem 5.5.2 (Persistency)

The interaction net rules are sound with respect to the path semantics; if $N \rightarrow N'$ is a valid interaction, then all paths in N survive the action of the cut.

PROOF: We first show the interactions for the PCF interaction from Figure 5.10.

1. $f \bowtie n$. There are four rules encoded here, and all are identical. We show the **iszero** $\bowtie n$ interaction, where $n \neq 0$.

Before the interaction we have only one path, which we reduce as follows:

$$\begin{aligned}
 !(\text{iszerod}^*)t^*!(n)t!(d) &= !(\text{iszerod}^*)!!(n)!(d) \\
 &= !(\text{iszerod}^*dn) \\
 &= !(\text{ff})
 \end{aligned}$$

After the interaction, again we have only one path $!(\text{ff})$, hence the path survived the action of the cut, and the paths are identical.

2. **cond** $\bowtie \text{tt}$. Before the interaction, the path construction is given by:

$$\begin{aligned}
t!((d\mathbf{ff}^* + d\mathbf{tt}^*)d^*)t^*(\mathbf{tt})t!(d) &= t!((d\mathbf{ff}^* + d\mathbf{tt}^*)d^*)!(\mathbf{tt})!(d) \\
&= t!((d\mathbf{ff}^* + d\mathbf{tt}^*)\mathbf{tt}) \\
&= t!(d)
\end{aligned}$$

After the interaction we are left with the path 1, and an inaccessible path on the false branch. Hence by familiar arguments, the path is preserved under reduction.

The case for **cond** $\bowtie$ **ff** is similar.

Next we show that the PCF rules are well behaved with the duplicator nodes, as given in Figure 5.11.

1. $1 \bowtie c$. Before the interaction there are two paths $r_k^*(c)r_k$ and $s_k^*(c)s_k$. Both these paths have weights $!(c)$, which is exactly the weight of both paths after the reduction.
2. $1 \bowtie f$. Before the interaction we have the path $!(fd^*)t^*r_k = r_k!(fd^*)t^*$ which is exactly the weight of the path after the interaction.
3. $1 \bowtie \mathbf{cond}$. Before the interaction, we have the path $t!(d\mathbf{ff}^* + d\mathbf{tt}^*)(d^*)t^*r_k!(\phi)r_k^*t!(d)$. After the interaction we have $r_k t!(d\mathbf{ff}^* + d\mathbf{tt}^*)(d^*)t^*(\phi)t!(d)r_k^*$. It is again straightforward to show that the path is preserved.

□

Since there is only one execution path for programs of ground type, this path survives the reductions and is the interpretation of the final result.

5.5.1 Further Extensions

With PCF we can capture all recursive programs hence, computationally, this is quite sufficient as a programming paradigm. However, practically we would not normally program, for example, addition as

$$Y(\lambda f.\lambda n.\lambda m.\mathbf{cond} \ (\mathbf{iszero} \ n) \ m \ (f(\mathbf{pred} \ n)(\mathbf{succ} \ m)))$$

but expect it to be built in.

For both efficiency and pragmatic reasons, we show very briefly how to add constants such as *addition* to this framework.

The basic arithmetic operations are very straightforward to code in an interaction net. Figure 5.13 shows the case for addition; the other operations follow the same principles.

The basic idea is to introduce a new node for each binary operator, for example $+$. An interaction with an integer on its principal port generates a new node $+n$ which is addition partially evaluated with some integer n . To complete the addition, we define a rule for the interaction of $+n$ and an integer m . Note that we are doing nothing more than defining how to do currying in an interaction net framework. We show in Figure 5.13 an example of these interactions.

We leave the interested reader to generate the additional rules for the interaction of these new nodes with contraction and weakening nodes, which are straightforward to generate.

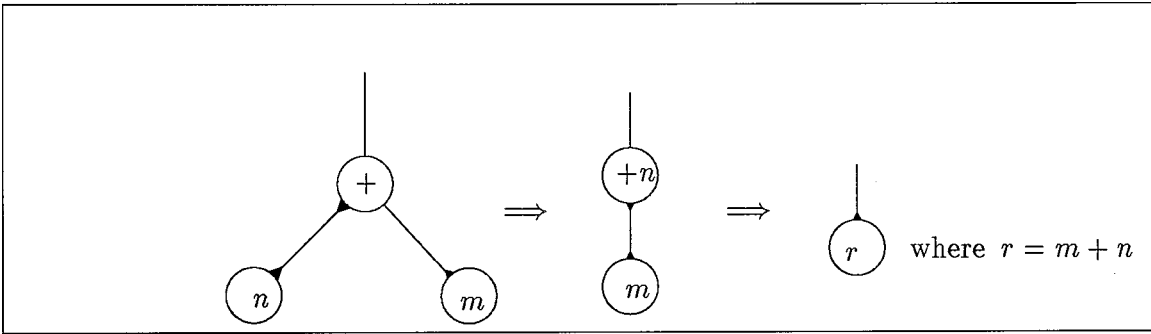


Figure 5.13: Extensions for arithmetic operations

General recursive Data-type such as lists, trees, etc. can also be handled in an interaction net framework. Here we just hint at how this can be done and leave it to further work for a full treatment.

We will show how to add a list data structure, which is the simplest example to show the general principles.

We introduce two nodes¹ for the list data type constructors **nil** and **::** (cons):



Next, there is a need to introduce functions which work over this data type, for example **isnil**, **hd**, **tl**, etc., and show how these nodes interact with the list data structure. We show the interactions for some of these functions, and leave the reader

¹We could code the list data type using existing constructors from Linear Logic, following the equation $\text{list } A = 1 \oplus A \otimes \text{list } A$, however we prefer to introduce them directly.

to complete the presentation for the rules of weakening and contraction acting on these agents. These interactions are shown in Figure 5.14.

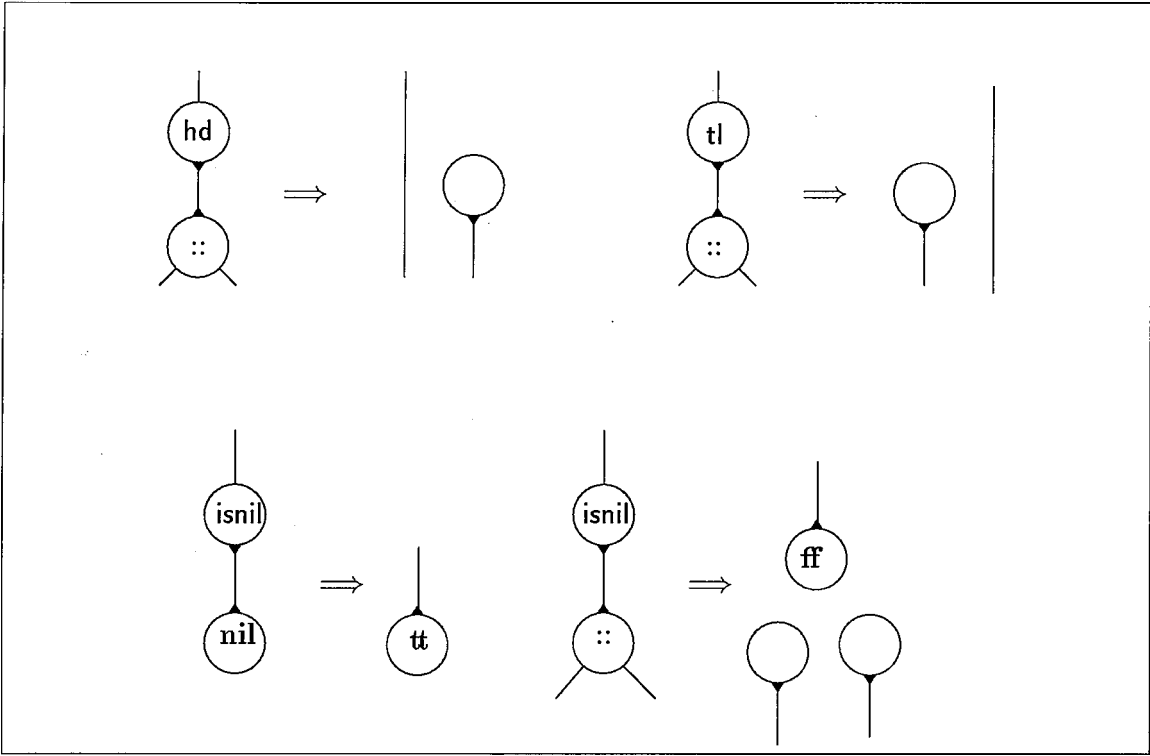
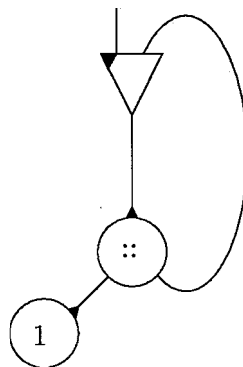


Figure 5.14: List Interactions

As usual, we can label the ports of the nodes with types, and show properties analogous to what we showed for PCF. There is also an extension of the dynamic algebra and context semantics that we can use to show an invariance of the rewrite rules. We postpone these ideas until the next chapter where we will briefly look at implementing the path computation associated with these kind of data types.

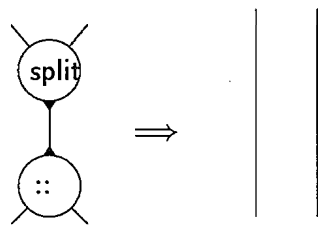
There is also a simple way for encoding cyclic data structures in this framework which is the same as the presentation of recursion. The following diagram is shown

as an example of how to code an infinite list of “ones”.



More generally, *process networks* [FH88] can be coded.

The above presentation of lists is by no means the only possible one, and there are many other functions that we could define. Again, just to give a hint of some of these ideas, note that the rules for `hd` and `tl` delete the part of the list that is not used. There may be situations when we reuse both components, and rather than copy the structure, and delete the head of one list and the tail of the other, we can define directly a function that will return both the head and tail of the list; i.e. `split` the list. These kind of operators have previously been studied in the framework of Linear Logic calculi by Holmström [Hol88], and also in [Mac94]. We just show the interaction of a `split` node against a list.



5.6 Analysis

We stress that we do *not* get optimal reduction in the sense of Lévy’s criterion. We must therefore try to characterise exactly what sort of sharing we are capturing.

The most elegant way of showing that an algorithm of this kind is optimal is to use a notion of labelling, introduced by Lévy [Lev78], which is a technical tool ideal for the study of *residuals* and optimal reduction in general. This technique has been used most successfully, amongst others, in [GAL92].

Optimal reduction says that we should *never* have two edges in our graph having

the same label. We show where in our algorithm we fail to achieve this. Consider the following example:

$$(\lambda x.xx)((\lambda xy.xy)(\lambda x.x))$$

The graph representation is shown in Figure 5.15, together with the residual graph after 2 interactions.

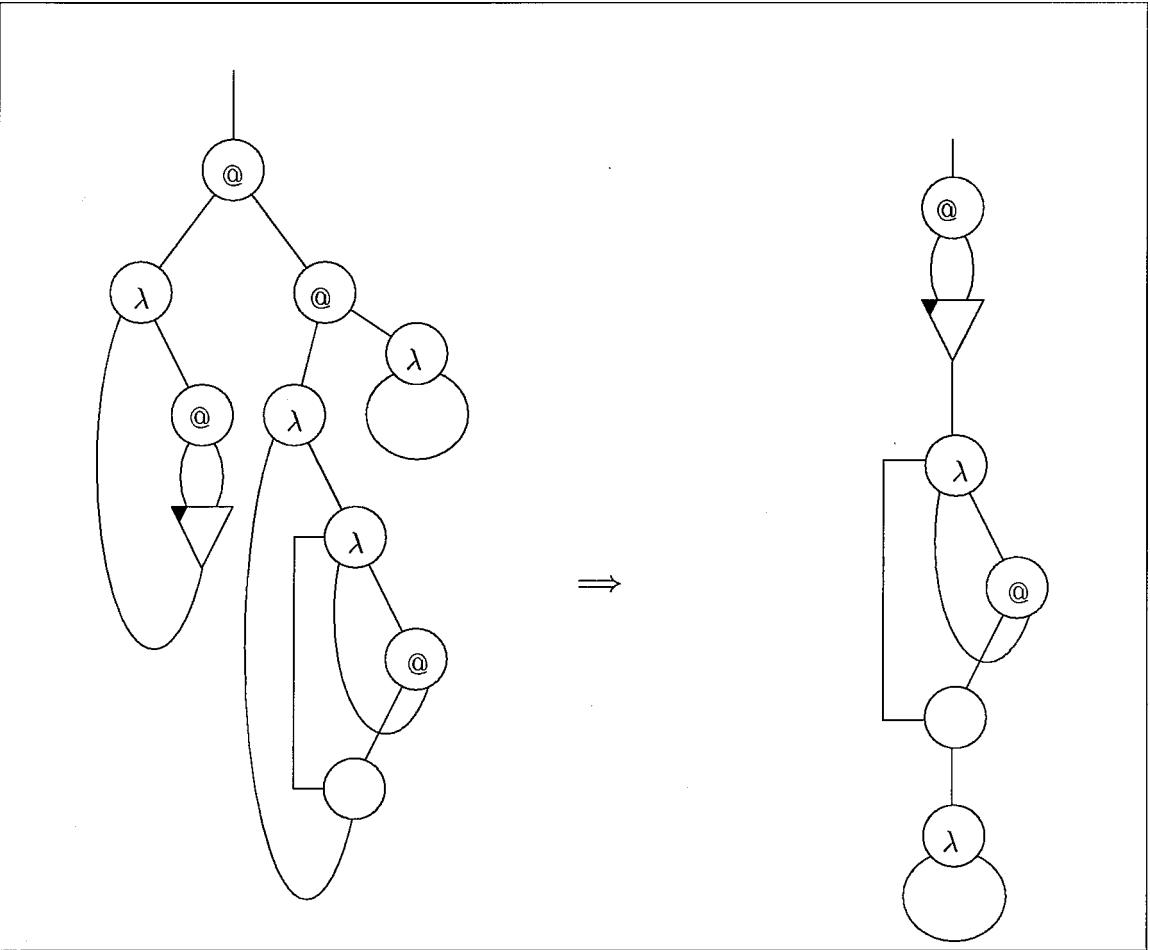


Figure 5.15: A counter example for optimality

Since we do not do commuting conversions we cannot push the substitution $\lambda x.x$ into the term. Hence only the fan-lambda interaction is possible, which will cause the duplication of the lambda term and its substitution, i.e. the redex $(\lambda x.x)y$ will be copied.

Results are showing that we can actually do less work; where work is defined to be the number of interactions performed. We accept the scepticism that could be brought up with respect to this measure of work in particular, counting the number of interactions does not tell us anything about parallelism.

It is easy to find an example where our algorithm performs significantly better than extant optimal interpreters. Consider the term:

$$(\lambda x.\lambda y.xy)BIG\ M$$

where *BIG* is a term with a normal form of considerable size; for example a large Church numeral.

The graph representation of this term is shown in Figure 5.16, where one reduction is shown. A comparison of *what happens next* shows where the gain is achieved.

- In our algorithm the top redex will fire, deleting the box structure. This causes the term *BIG* to become aligned with the application so further interactions are possible.
- With the optimal reducers this same interaction is possible, but since there is no method of erasing the box structure, the side doors of the box stay in the network. This causes the term *big* to be *lifted* into the box through the side door. The cost of fully lifting the term *BIG* into the box is the size of the term in normal form, *i.e.* the number of interactions required depends on the size of the term.

Here, we just mention a couple of points to give some idea of how much less work we do than [GAL92].

1. With respect to Lamping's quote given in the introduction, we point out the most fundamental gains and losses of our algorithm.

Since we only push substitutions in when the !-box is opened (by an application), the substitution moves in for free. In particular, the cost does *not* depend on the size of the term being substituted in, as in [GAL92] where the cost in terms of the number of interactions is *n* times the size of the term (in normal form) being substituted in where *n* is the number of accumulated bracket and croissant nodes (see below).

Our loss is that we may copy this substitution—see the example above.

2. If a term is syntactically linear, then our algorithm is optimal in the sense of Lévy's criterion. Moreover, we do far less work in terms of number of interactions.
3. No accumulation of bookkeeping nodes. This is regarded as the main inefficiency of optimal reducers and has been studied by Asperti and Dore [AD94]. With

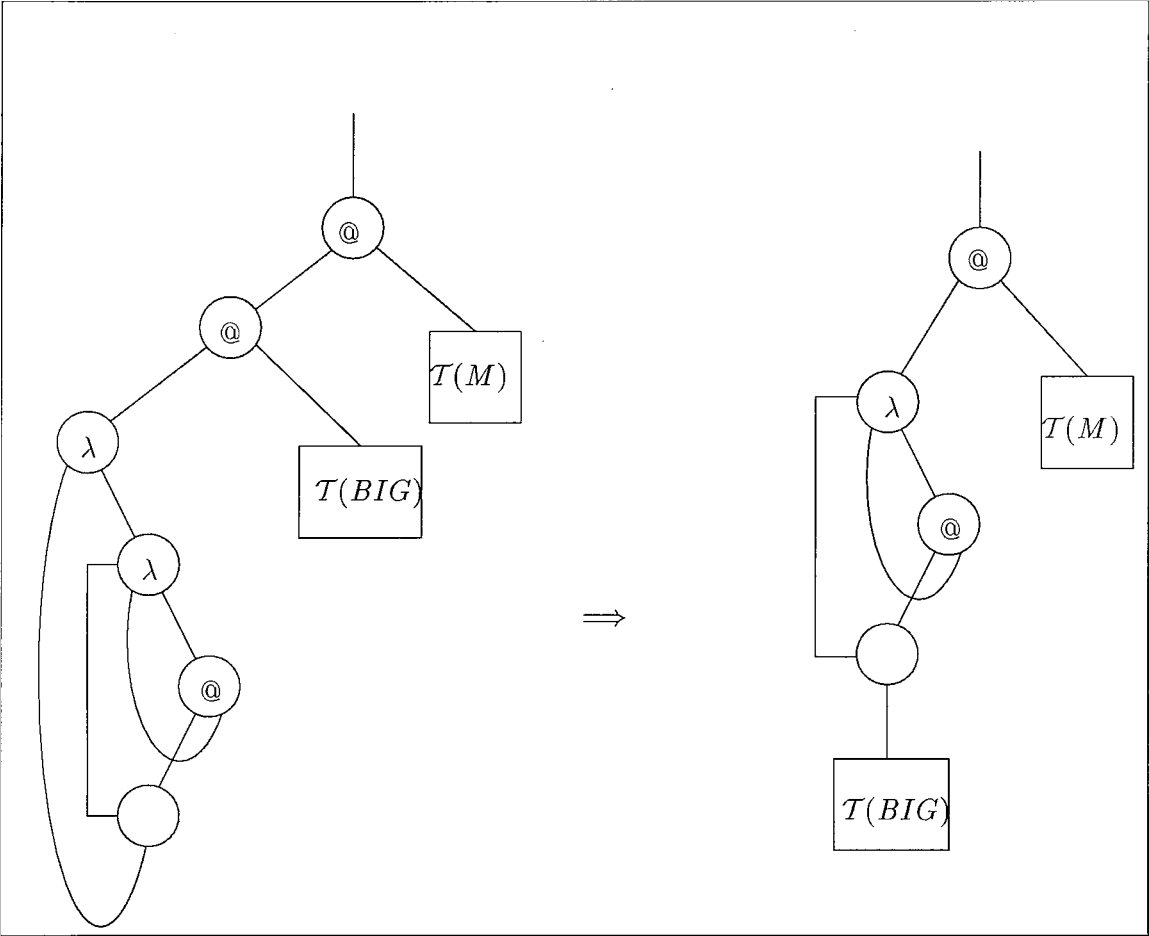


Figure 5.16: An example of how our algorithm gains

our implementation we in fact have *no bookkeeping* interactions. All interactions are required as part of the reduction; there are no interactions that are taking place just to change indexes on control nodes. The accumulation of bookkeeping nodes is in fact related to the problem of the “missing” $t!(d) = 1$ equation in the Geometry of Interaction, which we discussed in Chapter 4.

5.6.1 GOI-Tools

To obtain a global understanding of this algorithm, an implementation has been developed. This is part of a much larger development system that was designed to investigate and compare various interaction net implementations (and, as we will see in the next chapter, implementations of the Geometry of Interaction; hence its name) and is reported in Appendix A.

The motivations for the implementation were to reassure us that the ideas worked and obtain benchmark results. Within the interaction net framework there is an evident notion of *work* required during a reduction which is simply based on counting the number of interactions.

Benchmark Results

To justify our claim that we perform less interactions, we show some benchmark results. These results were obtained using GOI-tools for our algorithm, and Version 0.0 of the Asperti and Laneve optimal interpreter.

Since both interpreters perform reduction only to some weak notion of normal form, we ensure that our examples are provided with sufficient arguments so that reduction to a full normal form in both cases is achieved.

A selection of terms used in comparisons are shown below, where the input to the Optimal Interpreter had the control information of the Linear $\lambda\sigma$ -calculus stripped out.

Church numerals:

$$\begin{aligned}\bar{2} &= \lambda f.\lambda x.[f = \langle f_1, f_2 \rangle]f_1(f_2x) \\ \bar{3} &= \lambda f.\lambda x.[f = \langle f_1, f_2 \rangle][f_2 = \langle f_3, f_4 \rangle]f_1(f_3(f_4x)) \\ &\text{etc.}\end{aligned}$$

Lamping’s example:

$$L = (\lambda g.[g = \langle a, b \rangle](a(bI)))(\lambda h.(\lambda f.[f = \langle c, d \rangle](c(dI)))(hI))$$

Lévy’s example:

Term	GOI-Tools		Optimal Interpreter	
	Interactions	beta	Interactions	beta
III	2	2	18	2
$\overline{22}II$	33	11	204	9
$\overline{33}II$	221	45	652	15
$\overline{222}II$	192	42	786	16
L	25	10	179	10
LI	26	11	192	11
J	7	5	59	5

Table 5.1: Benchmark Results: Interaction nets

$$J = (\lambda x.xI)(\lambda y.(\lambda z.[z = \langle u, v \rangle]uv)(yI))$$

The results are given in Table 5.1, where we show the total number of interactions performed by each evaluator, and also show how many of the total number were β reductions.

We hinted earlier that this algorithm relates strongly with Wadsworth’s algorithm for Graph reduction. Here we mention a few points to show this, and also indicate where we do better. We refer the reader to the discussion of Wadsworth’s algorithm in Chapter 2 to aid reading this section.

Recall that there were three main features of Wadsworth’s algorithm.

- Overwrite root to share redex.

This is implicit in our implementation since all operations are performed in-place.

- Copy λ -body to avoid substituting wrong variable. This is achieved by our propagation of the duplicator nodes. Note in particular that we contract all redexes in the body while making the copy. This in fact corresponds to performing λ -lifting; we identify *maximal free expressions* at run time.

In addition, we *only* copy the λ -body if we really need to.

- Substitute pointers to avoid excess copying. All our rewrite rules are implemented by simple *pointer switching*. In addition, bound variables point to the binding lambda so we do not need to traverse the structure of the term to do substitutions of pointers of formal parameters to actual parameters.

We conclude this section with a few additional features of this implementation.

- Garbage is clearly identified in this framework. Sub-nets become detached from the structure by the use of the weakening nodes introduced. These indicate exactly which parts of the net need to be erased since we know the extent of the box.
- We indicated above that there were no additional “bookkeeping” interactions required in this interaction net. The only interactions that are not part of computing the result directly are the copying (duplicator) interactions. However we can show that in fact we do the least amount of work possible to perform the copying:

Proposition 5.6.1 Let M be a term in restricted substitution normal form, then the number of interactions required to copy M is exactly $|M|$, the size of M . $\diamond$

5.7 Alternative reductions

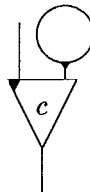
Non-Interaction Nets

There are no claims in this thesis that we have produced the “best” interaction net implementation of the λ -calculus. Here we give some indication on how we might further improve the situation. These are given without proof of correctness or detailed analyses of cost of reduction.

There are several inadequacies of the implementation that we were unable to resolve simply because we have restricted ourselves to an interaction net paradigm. Here we will look at two specific, frequently arising situations that would make the implementation even more practical. We will look at each of these ideas in turn.

Contraction/Weakening

In all the interaction net implementations of Linear Logic, there is an unfortunate configuration of agents which in fact causes a great deal of additional work in reducing the term to normal form. Consider the following example $(\lambda x.(\lambda y.I)xx)$. After one interaction there is an unfortunate agent configuration:



This is a situation arising from a proof consisting of a contraction followed by a weakening, *i.e.* we copy part of the proof structure, then erase one of the copies. This situation is frequently generated during cut elimination.

In the world of interaction nets, there is no way that we can eliminate this into a single edge, however, as is shown in [Bec92] the reduction is valid. It is also the case that the accessible paths in the term are the same in the Geometry of Interaction interpretation, *i.e.* they are persistent.

An implementation of our rewrite rules could add this rule as some kind of meta-rule. We can perform this reduction whenever we see it, and we will not affect the properties of the interaction net system. In particular, confluence and normalisation are preserved.

Commutative conversions

The only reason that our implementation fails to achieve optimal reduction is simply because we do not perform the commutative conversion (pushing in a substitution). Again, we can escape from the pure world of interaction nets and have this as a meta-rule. Note that by the “well balanced” condition we do not have any problems of conflicting fan nodes.

The meta-rule would thus be as shown in Figure 5.17. Although this operation looks rather complicated, it can in fact be implemented in a very simple way by pointer manipulations.

A sequence of commutative conversions gives rise to an associativity problem. Consider the following example: $!(P) \cdot_x !(Q) \cdot_y !(R)$ with $x \in fn(P)$, and $y \in fn(Q)$. There are two ways in which the commutative conversions can take place.

First, using a leftmost reduction we have:

$$\begin{aligned} !(P) \cdot_x !(Q) \cdot_y !(R) &= !(P \cdot_x !(Q)) \cdot_y !(R) \\ &= !(P \cdot_x !(Q) \cdot_y !(R)) \\ &= !(P \cdot_x !(Q \cdot_y !(R))) \end{aligned}$$

and, using a rightmost reduction, we have

$$\begin{aligned} !(P) \cdot_x !(Q) \cdot_y !(R) &= !(P) \cdot_x !(Q \cdot_y !(R)) \\ &= !(P \cdot_x !(Q \cdot_y !(R))) \end{aligned}$$

Obviously, the second reduction is shorter — one can see that by doing the rightmost substitution first, the $!(R)$ gets a “free ride” into the boxed $!(P)$. Hence an *inside-out* reduction schema provides a more efficient implementation of commutative conversions.

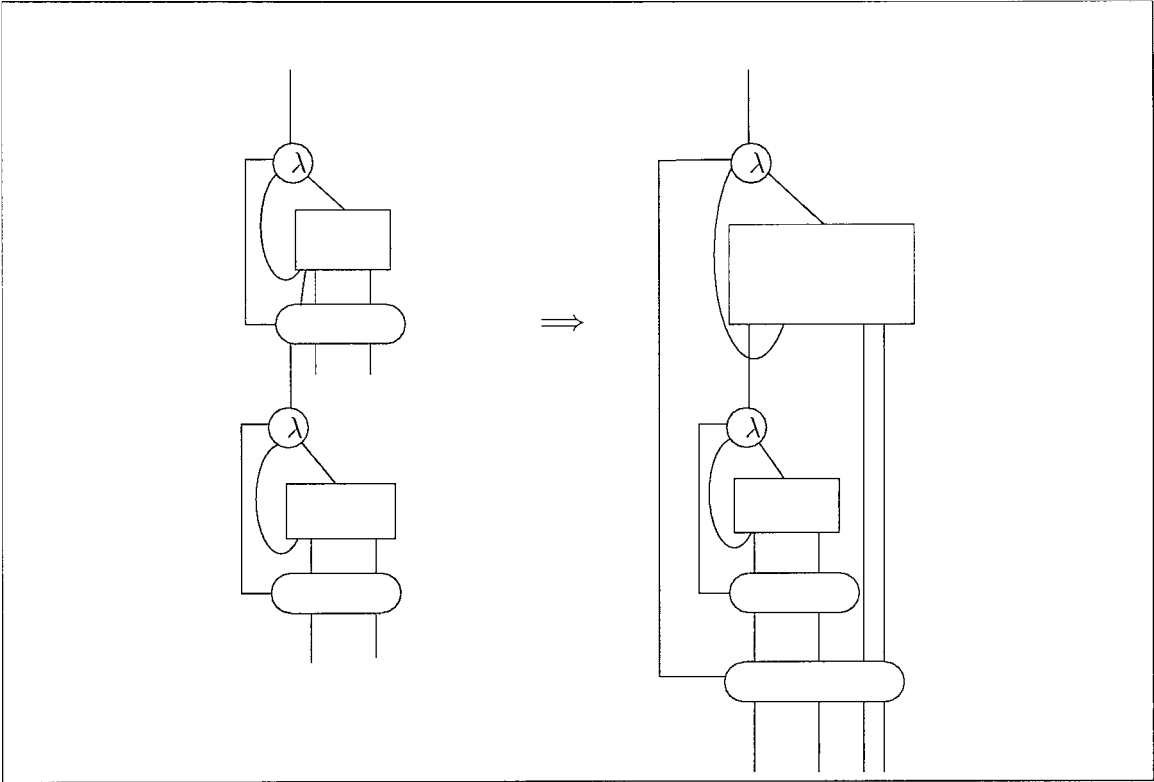


Figure 5.17: Commutative Conversion

5.8 Discussion

In this chapter we have presented a very simple interaction net implementation for PCF. Although not optimal, we have incorporated some of the notions of partial sharing that arose naturally from looking for an interaction net for Linear Logic proof structures.

The implementation seems to perform reasonably well on a prototype implementation, and we feel that there is potential for such an implementation for a realistic functional language. What remains however is the problem of successfully implementing interaction nets. There have been several implementations, but nothing has yet emerged beyond a prototype form. In particular, they have *trivial* parallel implementations, but to our knowledge, not a single one has been developed. With the development of such a platform we will be in a better position to declare the usefulness of this approach.

There are many other interaction net implementations of the λ -calculus. The one discussed was a very simple system, and provided a good starting point for further work. An important point to note about these Interaction Net implementations is that everything is made explicit—copying and discarding. These (expensive) operations are usually hidden. Therefore we can analyse the *full* cost of evaluation in a unified framework.

We chose to use a specific translation of the λ -calculus into Linear Logic proof structures for our presentation here. This was purely because we did not want to present both. The details of the other translations have equally well been worked out, and all the properties go through without problem.

More generally, we followed the philosophy of defining agents so that we could translate PCF into interaction nets. Of course, this is not the only way of doing things. In particular, one could program directly in interaction nets; Lafont has defined a language for doing this. We just remark here that the way we did things is not the only way.

Chapter 6

A Sequential Data Flow Machine

This chapter gives our second major contribution to the thesis. We consider the proof technique used in Chapter 5 as an implementation technique directly, yielding the Geometry of Interaction Machine.

6.1 Introduction

In this chapter we will see how one can use the path semantics (used as a proof technique in the previous chapter) in a more direct way to give a quite different implementation.

The implementation paradigm considered here is that of *data-flow*; more specifically, pushing a *single* token around a fixed network.

- The network is a Linear Logic proof structure generated by the given $D = !(D \multimap D)$ translation from the λ -calculus, and extended to cover the constants of PCF; so we are working in a typed setting.
- The token is the *context* data structure $(\mathcal{M}, \mathcal{E}, \mathcal{D})$ that we used as a (small) model of the dynamic algebra Λ_{pcf}^* .

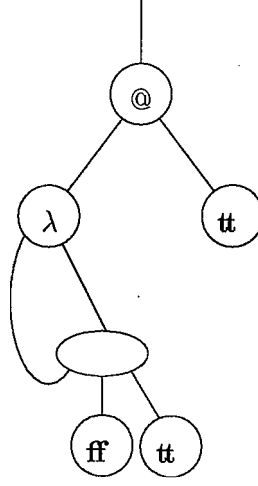
To motivate our ideas, and to compare with the work of the previous chapter, we show an example of what we are trying to achieve. Consider the following PCF program, which is an application of the negation function:

$$(\lambda x. \text{cond } x \text{ ff tt}) \text{tt}$$

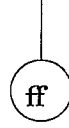
In our previous work there were three distinct steps necessary to produce the result of the computation:

1. Compilation into interaction net structures. Thus the above example would

be encoded as the following net using the compilation function defined in Chapter 5:



2. Evaluation to normal form using the dynamics of the interaction net. After three interactions (including one garbage collection), we arrive at the normal form of:



3. Reading back the answer of the computation from the resulting interaction net, we get the PCF constant **ff**.

Our approach to computation here is somewhat different, in that we want to try to do the final phase of the above *without* reducing the net to normal form. Hence we begin by coding the term into a net structure as above, then we extract the result using the Geometry of Interaction; we calculate the *execution path* starting at the *root* of the term. To push this example to the end, we show the path computation:

$$q^* d^* !(q!(d^*)t!(\mathbf{ff})t!(d\mathbf{tt}^*d^*)t^*p^*)dp!(\mathbf{tt})p^*d^*(pt!(d)q^*)dq$$

which has weight $!(\mathbf{ff})$ as required.

Of course, we now have to show how to construct this path at run time, and devise a way of implementing the equations of the algebra Λ_{pcf}^* . We will use the ideas presented in Chapter 4 where we first showed how to construct the execution path by starting at the root, and then showed how the context semantics can be used to

model the dynamic algebra. Hence, our token travelling around the network will be the context data structure $(\mathcal{M}, \mathcal{E}, \mathcal{D})$, (which one can see as a global *state* of the computation), which will be transformed by the operators of Λ_{pcf}^* .

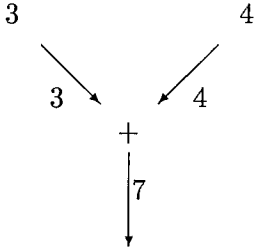
For the purpose of this chapter we will work with evaluating *programs* (closed terms) at *ground type*. The immediate consequence of this is that there is a *single* (unique) root-to-root execution path. Using a result from Chapter 4 (Proposition 4.5.10) we know that travel of the token is *deterministic* — at each choice point in the travel the token always has sufficient information to decide which way to go next. Of course, this restriction to ground types is no more than a design decision. We justify not evaluating programs of higher types by the principle that the result of a computation should be observable data. A functional result can only be observed by evaluating on an argument, and as far as the user is concerned (s)he will not be able to observe the difference between evaluation to some notion of normal form and non-evaluation. As an aside we remark that an alternative philosophy would be to suspend the computation for a higher order computation and prompt the user for further input so that the computation can proceed. This is particularly of interest for *lazy* implementations where we would like to experiment under the head lambda to enable the user to distinguish between different terms.

We will first review the notion of data-flow that we will be considering and show how this is nothing more than the path semantics given in Chapter 4. We then give a concrete implementation of this data-flow directly in *assembly language* for an abstract machine. Finally, we take a detailed look at some optimisations that can be performed and some variants on the ideas presented.

6.2 Sequential Data Flow

The theory for this chapter has been developed in the preceeding chapters. All we do here is put a new perspective on this theory to develop a concrete computational interpretation of Girard's Geometry of Interaction, extended to cover the PCF constants.

In this section we review the notion of data-flow that we will be using. Let us begin by stating what is *not*. The general view of data-flow is that of Kahn [Kah74] which can be summarised as a network which is a directed graph with nodes of the graph being autonomous computing agents. The edges are *unidirectional* unbounded FIFO queues; so they are buffers. For example, consider the following very simple data flow network which we will explain in the framework of *op. cit.* and then show how our ideas will work.



The idea is that the constants 3 and 4 will generate the value and place it on its output queue. The addition node (+) can *fire* when both inputs are available, and then perform the addition which is then put on the output queue as shown above.

The kind of data flow that we have in mind is quite different. We traverse the structure from the root, and try to build up the result in the token. Each node will have a “redirection” rule which will tell the token which way to go next (maybe by looking at the structure of the token). So, looking at the example above, we begin at the root, and reach the addition node. This node will direct the token to one of its arguments (say for example the left argument). The token then travels to the 3, at which point we store the value in the token (say in a stack), and return back along the edge to the + node again. We now interrogate the right argument, which will lead us to storing the value 4 in the token stack. Again we return to the + node, but this time, exit back to the root and applying the addition function to the two values in the token. Hence, we have traversed the graph in a very standard way, but with a data flow perspective.

We point out some of the significant differences:

1. The edges of the network are *bidirectional*; the token will travel in both directions along an edge.
2. There is a single token travelling in the network at any point during the computation.
3. There is no notion of *buffering* on the edges — the main implementation difficulty of standard data-flow.

The Geometry of Interaction allows this simple idea to be extended to the λ -calculus and using our extensions we can code PCF and other constants typical in functional programming languages.

6.3 The Geometry of Interaction Machine

In this section we will actually provide a computational interpretation of Girard's Geometry of Interaction by providing a compilation directly into assembly language. We will provide a compilation module for each logical connective and PCF constant of our language.

We begin by declaring our object language which we take to be a very simple assembly language.

We will assume a simple register machine with, say, 32 bit (unsigned) registers $R0$, $R1$ etc. The operations over these registers will be simple logical operations (such as shifts), comparisons and branching instructions. We propose a basic set of instructions which have trivial implementations on any architecture. The following table gives the intended meaning of the instructions that we require to code the multiplicative and constant information. We will introduce the additional structure for the exponentials on demand. Additionally, we will define several *macros* to give greater abstraction for the compilation.

Macro	Instruction	Meaning
p	lsl R0	logical shift left R0
q	lsl R0	logical shift left R0
	inc R0	increment R0
	lsr Rn	logical shift right Rn
	mov n R1	move value n into R1
	br l	branch always to l
	be l	branch to l if carry equal to zero
	cmp0 Rn	compare Rn with zero
	inc Rn	increment Rn
	dec Rn	decrement Rn

6.3.1 Identities

The compilation of the identities of Linear Logic proof structures produces no code, they just provide *linking information* to show how nodes are connected.

Axiom: The axiom is given by the identity on contexts, hence no context transformation is required. For our implementation, the axiom simply provides a piece of linking information to state which nodes are to be connected.

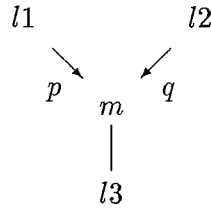
Cut: Again, no context transformation is required. The cut link specifies which nodes should be cut against each other, hence again we just provide linking

information here.

6.3.2 The Multiplicatives

The multiplicatives are straightforward to implement. By Definition 4.6.3 we have the fact that we can use a very small run-time system. We will show in fact that a *single* register will suffice.

We have seen how to assign paths to proof structures in Chapter 4, and we have also seen that the context semantics are a (small) model of paths, in the sense that we can see each node in a proof structure as a set of context transformers. For a multiplicative node we have three possible entry points (cf. the construction of a path):



1. If we arrive from the left hypothesis ($l1$), with context M , we pass control to the conclusion and record in the multiplicative stack the information that we came from the left: $l : M$.

This indicates a very simple compilation directly into assembly language, where we can use a single, “potentially infinite” register R0 as the multiplicative register. Using 0 as the coding of “left” we can simply compile this as:

```
l1 :  p
      br l3_out
```

Where `l3_out` is the *other end* of the edge $l3$, which will be an input to another node.

2. The token M arriving from the right hypothesis $l2$ requires that we record the information that we came from the right: $r : M$. Similarly to above, we can compile this as follows using 1 as the encoding of “right”:

```
l2 :  q
      br l3_out
```

3. Finally, arriving from the conclusion, we examine the top element of the multiplicative stack and branch to either the left or right premise, and drop the top element of the stack M . This gets a straightforward compilation as:

```
l3 : lsr R0
      be l1_out
      br l2_out
```

Hence, for each node in a Multiplicative Linear Logic proof structure we have a very trivial encoding. Each node is translated in to a module of assembly language code with three entry points. The compiler connects all labels which is the linking phase specified by the Axioms and Cuts of the proof.

l1 : p br l3_out	l2 : q br l3_out	l3 : lsr R0 be l1_out br l2_out
---------------------	---------------------	---------------------------------------

With these very simple rules we have the power to implement the linear λ -calculus, with *just one register run time system*. The size of the object code (number of lines of assembly language) is just eight times the number of constructors in the original program. This is not a very powerful system, but already a number of very interesting points arise here.

We suggested that *R0* is a *potentially* infinite register in the sense that in general we have no idea just how large this stack will grow. However, we are saved in a typed setting by the results in Chapter 4 — the size of the multiplicative stack is bounded by the size of the type of the subterms. Here we will assume that no program ever goes beyond a 32 bit register, which will allow the coding of programs at very *high* types. Of course, if we needed higher types, then we could extend the above encoding to handle larger stacks in a very simple way. Our aim here is to try to show just how simply we can code a PCF program.

6.3.3 The Exponentials

For the exponentials we are not so lucky in that we require a data structure a little more complicated than a simple register. We will assume a tree data structure and define operations on it corresponding to the exponential context semantics. We will define the operations as we go along and will refrain from giving a sequence of assembly language instructions that the interested reader will be able to construct for himself. We will however show the operations on an Abstract Data Type to make our presentation precise.

We require a binary tree, in which we can move around freely. For this it is proposed that a double linked tree be used; storing a pointer to the parent node. The context data type is defined by the following code that is presented using C syntax [KR88].

```
typedef enum { C_empty, C_cons } Ctype;
```

```
struct C_tree {
    Ctype type;
    integer value;
    struct C_tree *prev;
    struct C_tree *left;
    struct C_tree *right;
}

typedef struct C_tree CtreePtr;
typedef CtreePtr *Context;
```

We will assume a basic building operation on this structure which has the obvious behaviour:

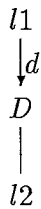
```
build : C_type -> Context -> Context -> Context -> Context
```

We keep a pointer into the structure to keep track of which part of the data structure we are using: the *environment pointer* *ep*. In other words, we have two pointers to the structure:

```
Context    context, ep;
```

Dereliction

Dereliction is the only connective that generates or destroys part of a context tree; hence it creates/removes levels of the environment.



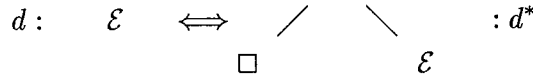
If we arrive from *l1* with exponential context tree $\mathcal{E}$ then we add a new level to the environment, hence create a new empty stack. We then continue downwards to the node connected to *l2*. Arriving from *l2* we do the reverse operation, and remove part of the environment. We should aim to handle the *d* operator in a careful way since we require dynamic memory allocation and garbage collection. The compilation rule for this node is given by:

l1: d br l2_out	l2: d* br l1_out
--------------------	---------------------

Where we can encode d and d^* by the following fragments of code:

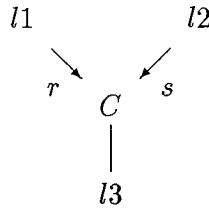
d :	Context $a = \text{build}(\text{ep} \rightarrow \text{type}, \text{ep} \rightarrow \text{left}, \text{ep} \rightarrow \text{right}, \text{ep});$ $\text{ep} \rightarrow \text{left} = \text{build}(\text{C_empty}, 0, 0, \text{ep});$ $\text{ep} \rightarrow \text{right} = a;$ $\text{ep} \rightarrow \text{type} = \text{C_cons};$ if $(a \rightarrow \text{left})$ $a \rightarrow \text{left} \rightarrow \text{prev} = a;$ if $(a \rightarrow \text{right})$ $a \rightarrow \text{right} \rightarrow \text{prev} = a;$
d^* :	Context $\text{tmp} = \text{ep} \rightarrow \text{right};$ $\text{ep} \rightarrow \text{type} = \text{tmp} \rightarrow \text{type};$ $\text{ep} \rightarrow \text{left} = \text{tmp} \rightarrow \text{left};$ $\text{ep} \rightarrow \text{right} = \text{tmp} \rightarrow \text{right};$ if $(\text{tmp} \rightarrow \text{left})$ $\text{tmp} \rightarrow \text{left} \rightarrow \text{prev} = \text{ep};$ if $(\text{tmp} \rightarrow \text{right})$ $\text{tmp} \rightarrow \text{right} \rightarrow \text{prev} = \text{ep};$

The effect of this code can be seen diagrammatically as the following context tree transformation:



Contraction

A contraction is entirely similar to the multiplicative node, except we must place the information on the current *fan stack*.



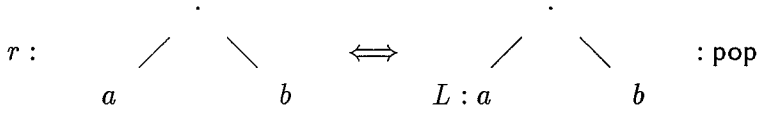
If we arrive from $l1$ (resp. $l2$) then we should record the information that we came from the left (resp. right) on the context tree, and continue towards the node connected to $l3$. Arriving from $l3$ will pop the current fan information from the context tree and branch to either $l1$ or $l2$ depending on the status of the context tree. Hence the compilation scheme for this node is given by:

$l1:$ r $\text{br } l3_out$	$l2:$ s $\text{br } l3_out$	$l3:$ pop $\text{be } l1_out$ $\text{br } l2_out$
-----------------------------------	-----------------------------------	--

where we can encode r and s by the following fragments of code:

$r:$	$\text{ptr} \rightarrow \text{left} \rightarrow \text{value} = \text{ptr} \rightarrow \text{left} \rightarrow \text{value} \ll 1;$ $\text{ptr} \rightarrow \text{left} \rightarrow \text{value} ++;$
$s:$	$\text{ptr} \rightarrow \text{left} \rightarrow \text{value} = \text{ptr} \rightarrow \text{left} \rightarrow \text{value} \ll 1;$

The following explains the action of the code on the context tree:



Finally, we give the code for `pop` which will remove the top element from the fan stack and set the carry flag accordingly.

<code>pop:</code>	<code>if (ep→left→value &1)</code> <code> carry = C_left;</code> <code> else carry = C_right;</code> <code> ep→left→value = ep→left→value >> 1;</code>
-------------------	--

Weakening

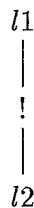
For a program of base type the root-to-root execution path will never arrive at a weakening node — it is not an accessible path from the root. For this reason no code is produced for a weakening node.

Promotion

As the token traverses the structure of the proof it will enter and leave ! boxes; changing the level of the environment that the token is in. There are two different cases to consider depending on how the token enters and leaves an exponential box. We will look at each one in turn.

Main Door

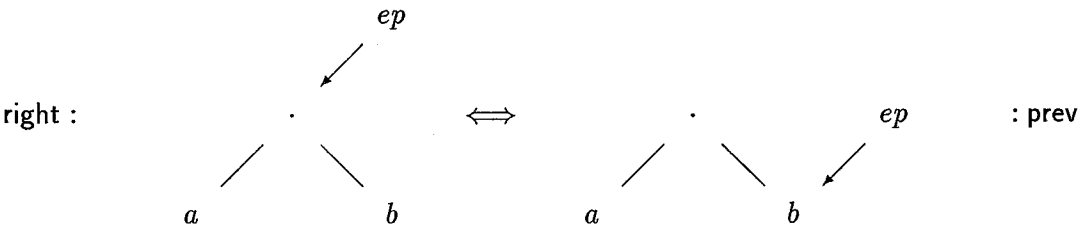
If we arrive at the principal port of a box then we increase the level of the environment that the token is working in.



The compilation of this idea can be given by the following:

l1: ep = ep→right br l2_out	l2: ep = ep→prev br l1_out
---------------------------------	--------------------------------

The effect of $ep = ep \rightarrow right$ and $ep = ep \rightarrow prev$ can be seen diagrammatically as the following *walk* on the context tree:



Auxiliary Doors

There are two steps involved in the token passing through an auxiliary door of a box.



- Analogously with passing through the main door, we must increase (if entering the box from $l2$) or decrease (if leaving the box from $l1$) the level of the environment that the token is operating at.
- If the token is leaving the box then the exponential context tree is transformed in such a way that the context of the box is saved somewhere safe so that it can be restored when the token re-enters the box. This is achieved using the t operator (associativity) which can be thought of as pushing the environment onto a stack for later use.

The dual of the above is performed if the token is entering the box from an auxiliary door.

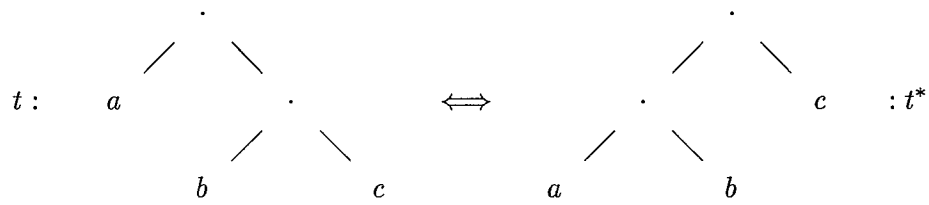
Hence the following code is compiled for each auxiliary door of a box:

l1: ep=ep→prev t br l2_out	l2: t* ep=ep→right br l1_out
-----------------------------------	-------------------------------------

A suggested coding of t and t^* is given by the following fragments of code to manipulate the context tree.

t :	<pre>integer b = ep→value; Ctype a = ep→type; Context tmp = ep→right; ep→right = tmp→right; tmp→right = tmp→left; tmp→left = ep→left; ep→left = tmp; ep→type = tmp→type; tmp→type = a; ep→value = tmp→value; tmp→value = b;</pre>	t^* :	<pre>integer b = ep→value; Ctype a = ep→type; Context tmp = ep→left; ep→left = tmp→left; tmp→left = tmp→right; tmp→right = ep→right; ep→right = tmp; ep→type = tmp→type; tmp→type = a; ep→value = tmp→value; tmp→value = b;</pre>
-------	---	---------	---

Diagrammatically this code transforms the exponential context tree as indicated below:



6.3.4 PCF constants

Here the extensions for PCF will be presented. The code generated for each element of Λ_{pcf}^* will be implemented using a single register $R1$ which will hold the result of the computation (cf. Definition 4.3.4). The compilation that we give here is not specific to any particular translation of PCF into proof structures. We leave the reader to construct the variants for each translation.

Constants at base type

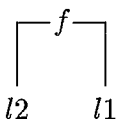
Both Natural number and Boolean values are straightforward to code — we just return along the same path having pushed the value onto the data part of the context, which again we can see as being a single register, $R1$. We will code $\mathbf{tt}$ as 0 and $\mathbf{ff}$ as 1, and natural numbers are compiled trivially. We write c for a general data value.



The compilation is straightforward, yielding the following code module for a constant:

l:	mov c R1
	br l_out

Arithmetic functions The unary functions **succ**, **pred** and **iszero** are given by the following:



From *l1* we simply pass straight through without change to the context. On the return up through *l2* we apply the function to the data item. The compilation of this module can be given by:

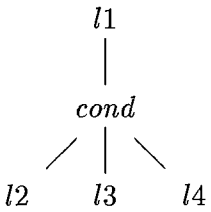
l1: br l2_out	l2: f
	br l1_out

Where the different functions (*f*) are coded as follows:

succ = inc R1
pred = dec R1
iszero = cmp 0 R1

Conditional

A conditional is slightly more complicated. We have the following module with four inputs/outputs.



When we ask for the result of a conditional we first ask for the boolean value, hence branch to *l2*. When we return, we either pass control to the true or false branch, depending on the answer in the register *R1*. Note that this data value is consumed by the test. The result of either branch is then passed back as the result of the conditional. This suggests the following code fragments:

l1: br l2_out	l2: lsr R1
	be l3_out
	br l4_out
l3: br l1_out	l4: br l1_out

Recursion

Our compilation of recursion introduces no new nodes, and hence does not require any further mention; it is dealt with by the compilation of the contraction and the application nodes.

6.3.5 Execution

The execution of a program is simply given by starting at the root of the term with the empty context $(\square, \langle \square, \square \rangle, \square)^1$ (so clearing R0, R1 and initialising the tree structure to be the empty tree) and following the path in the code generated. When the path reaches the root again we will have a token structure $(\square, \langle \square, \square \rangle, d)$ where d is the result of the computation (hence the result is in R1).

Example 6.3.1 We give a small example to show how things work, using the $!(A \multimap B)$ translation:

$$(\lambda x. \text{cond } (\text{iszero } x) \ 1 \ x)3$$

The execution path is given by:

$$q^*d^*!(q!(d^*)t^*s^*p^*)dp!(3)p^*d^*(pst(!d\mathbf{ff}^*d^*)t^*(\text{iszerod}^*)t^*r^*p^*)dp!(3)p^*d^*(prt!(d)t!(d)q^*)dq$$

which, using the equations in Section 4.2, reduces to $!(3)$ as required.

The output of our compilation algorithm is given in Figure 6.1. which, when run on our implementation, (with tracing set), produced the following output:

Finished after 58 stack operations

Result: 3

The implementation also produced the trace of context transformations given in Figure 6.2, and continued in Figure 6.3. The trace shows all 58 stack operations, which we name using the operations from Λ_{pcf}^* ; with the addition of using the notation $!$ and $!^*$ to indicate that we enter and leave a box respectively.

◇

¹Note that we set the empty context for the exponential component to be $\langle \square, \square \rangle$. This is required since values under this translation are represented as promoted values.

l011: q	br l1051
d	l642: t*
br l221	d*
l112: p	ep=ep→prev
d	br l422
br l221	l943: t*
l213: d*	d*
lsr R0	ep=ep→prev
be l181	br l422
br l000	l1044: t*
l221: ep=ep→right	d*
lsr R0	d
be l333	t
br l441	br l631
l422: q	d
ep=ep→prev	t
br l213	br l971
l323: p	l1051: br l732
ep=ep→prev	l1152: cmp 0 R1
br l213	br l1044
l631: s	l971: ep=ep→right
br l323	mov 1 R1
l732: r	ep=ep→prev
br l323	br l943
l333: pop	l181: ep=ep→right
be l1152	mov 3 R1
br l642	ep=ep→prev
l441: ep=ep→right	br l112
d	l000: end
t	

Figure 6.1: Object code produced

Extensions

Here we hint how to extend the ideas beyond PCF by showing how to code arithmetic operations and simple data structures.

Arithmetic

For arithmetic operations (+, −, *, div) there is a need for a *stack* of data values — it is no longer the case that a single data value will suffice.

We show the general case using addition as an example, leaving the reader to construct the additional cases for subtraction, multiplication and division.

First, we show the variant of the constants, which now are based on a stack.

$$\begin{aligned} \mathcal{D} &::= \square \mid d : \mathcal{D} \\ d &::= \mathbf{tt} \mid \mathbf{ff} \mid n \end{aligned}$$

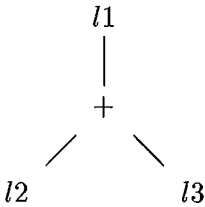
Hence, the abstract machine is extended with a data stack and corresponding operations as given in the following table:

Instruction	Interpretation
push <i>n</i>	$l \rightarrow n : l$
add	$x_1 : x_2 : l \rightarrow x_2 + x_1 : l$
mul	$x_1 : x_2 : l \rightarrow x_2 * x_1 : l$
sub	$x_1 : x_2 : l \rightarrow x_2 - x_1 : l$
div	$x_1 : x_2 : l \rightarrow x_2 \text{ div } x_1 : l$

The new coding of a constant is now given by pushing the value on the data stack, so the following code fragment is suggested:

```
l1: push n
    br l1_out
```

Addition is compiled as a fragment of code that will pick up the left argument first, and return with the value on the data stack. Computation proceeds by picking up the right hand argument, then performing the addition before the computation proceeds out of this construct.



The compilation is then given simply by:

l1: br l2_out	l2: br l3_out	l3: add br l1_out
---------------	---------------	----------------------

Lists

Finally in this section we remark that lists can also be implemented in this data-flow paradigm, and hence it seems that the extension to general recursive data-types is not so far away. This would then allow us to make serious comparisons with existing implementation techniques. However, we must leave this to future work.

6.4 Discussion of the implementation

We first remark on a few salient features of this implementation.

- The size of the object code is proportional to the size of the original program. We do not have any form of exponential growth in code size. In fact we can be quite precise in that the number of lines of code generated is linear in the size of the program — each term constructor gets encoded into no more than a few lines of assembly language.
- The runtime system is very small: The multiplicative and data information is held in 1 register each; the exponential information is the only part which requires dynamic allocation of memory for which we cannot calculate the bound. However, each level of the environment is only one register.

In Table 6.1 we give some benchmark results to indicate just what work is required for an evaluation. The numbers indicated show the number of *stack operations*. Each operator of Λ_{pcf}^* is regarded as a single operation, although in reality, as we have seen, some of these are actually coded as several instructions. However, these benchmark results give some idea of the order of magnitude.

The results are to show the inefficiencies of this kind of implementation. In particular, these are applications of functions of very high type.

We also include the code size of the program—the number of lines of assembly code—to give the reader some insight into how compact the compilation really is.

We will shortly look at ways of reducing the number of operations required.

Term	Stack operations	code size
$III\mathbf{tt}$	113	83
$\overline{22}III\mathbf{tt}$	1377	274
$\overline{33}III\mathbf{tt}$	8209	356
$\overline{222}III\mathbf{tt}$	18321	383

Table 6.1: Benchmark Results: Basic Geometry of Interaction Machine

Our implementation of a path computation was based on a small model of the dynamic algebra Λ_{pcf}^* , namely the context semantics, which is implemented as a simple tree data structure. Here we look at some other models of Λ_{pcf}^* to show that we may be able to take advantage of different types of computer architectures.

A three integer model

Here we present a different model of Λ_{pcf}^* which allows us to maintain a run-time system in three integers; one integer for each of Multiplicative stack, exponential tree and constant stack—keeping with our methodology of separating the multiplicative, exponential and data information.

Definition 6.4.1 Let $\lceil \cdot, \cdot \rceil$ be a bijection $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, for example: $\lceil m, n \rceil = 2^m(2n + 1) - 1$.

The action of the elements of Λ_{pcf}^* on $\mathbb{N}$ are defined as follows:

The Multiplicative and exponentials are identical to that shown in Definition 4.2.5.

PCF constants

The PCF data values are defined on $\mathbb{N}$, with the unary functions being defined as partial functions over $\mathbb{N} \rightarrow \mathbb{N}$.

$$\begin{array}{llll}
 \mathbf{tt}(0) & = & 0 & \mathbf{ff}(0) & = & 1 \\
 \mathbf{tt}^*(0) & = & 0 & \mathbf{ff}^*(1) & = & 0 \\
 \mathbf{pred}(0) & = & 0 & \mathbf{pred}(n+1) & = & n \\
 \mathbf{iszero}(0) & = & 0 & \mathbf{iszero}(n+1) & = & 1 \\
 n(0) & = & (n) & \mathbf{succ}(n) & = & n+1
 \end{array}$$

◇

Proposition 6.4.2 The above yields a sound model of Λ_{pcf}^* .

◇

PROOF: We need to show that the equations given in Definition 4.2.4 are sound in this model. The multiplicative and exponential fragment were shown in

Proposition 4.2.6, and the cases for the PCF constants are straightforward to check. $\square$

Implementation issues

Although we have shown that the computational content of Λ_{pcf}^* can be encoded as a very small run-time system, there is of course the problem of *unpacking* and *packing* the integers. In particular:

1. We need space to store the temporary values as the numbers are unpacked — there is no information about the size of the context, hence no information about how much store is needed to keep these values.
2. As a direct consequence of the above point, the computational cost of unpacking and packing integers using the bijection is unknown, and potentially quite large.

The packing and unpacking of an integer can be trivially implemented, storing results in registers and heap space.

With these points in mind, we look at a simpler model that allows us to offset some of the computational cost by increased storage space; a stack.

A stack model of Λ_{pcf}^*

For a stack based architecture, we propose a stack based model.

Definition 6.4.3 Again, the multiplicative and exponential information is decomposed, and it is assumed there is a bijection $\ulcorner \cdot, \cdot \urcorner : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ as before.

Multiplicatives

The operations p and q are defined on $\mathbb{N}$ exactly as in the previous model.

Exponentials

The exponential coefficients are modelled on a stack, (or list), of natural numbers. We define the operations on $\text{list}(\mathbb{N})$ as follows.

$$\begin{array}{llll} r(x_1 : l) & = & 2x_1 : l & s(x_1 : l) = 2x_1 + 1 : l \\ t(x_1 : x_2 : l) & = & \ulcorner x_1, x_2 \urcorner : l & d(l) = 0 : l \end{array}$$

With the exponential morphism being defined as:

$$!(f)(x_1 : l) = x_1 : f(l)$$

PCF constants

These are modelled on $\mathbb{N}$ exactly as in the previous model. $\diamond$

Proposition 6.4.4 The above yields a sound model of Λ_{pcf}^* . $\diamond$

PROOF: The proof proceeds as usual. Here we just show a few of the exponential cases.

$$\begin{aligned} !(f)t(x_1 : x_2 : l) &= !(f)(\ulcorner x_1, x_2 \urcorner : l) \\ &= \ulcorner x_1, x_2 \urcorner : f(l) \\ &= t(x_1 : x_2 : f(l)) \\ &= t!!(f)(x_1 : x_2 : l) \end{aligned}$$

$$\begin{aligned} !(f)d(l) &= !(f)(0 : l) \\ &= 0 : f(l) \\ &= df(l) \end{aligned}$$

□

Implementation issues

We can implement this model on a simple stack based architecture. The general idea is to use the stacks to store the values for the $!$ operations. We have two stacks, the context stack as described above, and a dump stack to store values². The exponential is now encoded as a *rotation* of stacks: we push the top of the context stack onto the dump, and apply the transformer to the context stack before rotating the stacks back. Hence there is a stack discipline for the storage of unpacked integers in the notation of the previous model.

To make this more precise, we show the rules for the principal and side doors of a box.

If we enter a $!$ from the top with context $(x : C, D)$ then we continue within the box with context $(C, x : D)$. The reverse operation is performed when the token passes a $!$ in the opposite direction.

For a side door, we have $(x_1 : C, x_2 : D)$. We leave the box and then perform the t operation: $(\ulcorner x_2, x_1 \urcorner : C, D)$.

6.4.1 Optimisations

We now look at some negative aspects of these implementations, and look at ways in which we can improve matters.

²A very similar algorithm based on two stacks has been developed by Laurent Regnier [Personal communication] to compute well balanced paths.

The Geometry of Interaction can be regarded as being *locally* very efficient — it has provided a decomposition of the λ -calculus to a level where we can directly implement it on a concrete machine at the lowest level of detail. However, this is offset by the *global* performance. From Chapter 4 (Definition 4.6.3) we have that the length of a path may grow exponentially. We now go on to look at ways of reducing this bound.

Before we suggest methods to reduce the length of a path, we state some important properties about the length of the execution path. As stated, the path is *worst case* exponential in the number of cut elimination steps. We stress that this is the worst case, and this is reached only when we are dealing with terms of higher type. Most programs rarely go above second order, and one has to try very hard indeed to write programs of higher types. Hence we propose that in general this bound is never reached. This phenomenon is by no means unique in computer science; the exact same phenomenon arises in Milner’s type reconstruction algorithm for simple types. It is certainly well acknowledged that type reconstruction is useful and the worst cases simply don’t arise in practice. Another example of this phenomena is the Simplex algorithm.

We could content ourselves with this situation, but in fact there are several *optimisations* that we can supply that arise directly from properties of paths that we looked at in Chapter 4 that give substantial increase in performance.

We look at two kinds of optimisations that we can include into the Geometry of Interaction Machine. First we look at some *local*, or *peephole*, optimisations, then go on to more interesting global optimisations.

6.4.2 Local Optimisations

Here we consider some very basic “peephole” optimisations that can be performed on the code generated.

Our first observation is that many of the jumps that we perform are indirected via another branch; this is particularly true for the coding of the conditional for example. These can simply be compiled out.

There are some interesting local optimisations that can be done by looking at certain combinations of operations in Λ_{pcf}^* that arise in computations. To introduce an important one we first take a detour and look at a restricted form of exponentials in the logical system TLL — a version of Linear Logic where the exponentials have similar rules to that of the Model logic T.

Modal T Linear Logic (TLL)³

TLL is a restricted version of Linear Logic which arises by replacing the rule for promotion:

$$\frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A}$$

by the weaker, so-called *functorial*, version:

$$\frac{\vdash \Gamma, A}{\vdash ?\Gamma, !A}$$

First, note that this new rule is derivable in Linear Logic by a sequence of Derelictions on the context:

$$\frac{\vdash \Gamma, A}{\vdash ?\Gamma, A} \quad \vdots \quad \frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A}$$

Linear logic is recoverable by adding the following rule:

$$\frac{\vdash \Gamma, ??A}{\vdash \Gamma, ?A} \text{ tLL}$$

It is worth pointing out that the addition of this rule gives equivalence of Linear Logic up to provability, but *not* equivalence on proofs and cut-elimination. Additionally, this rule most certainly does *not* have the sub-formula property. However these remarks are of no interest to our present work since we are going to try to work without rule tLL and see what benefits can be obtained.

It is our claim that we can produce a much simpler data-flow machine. We first justify this, then go on to try to show what class of functions we can capture in this logic.

The first point to be made is the simpler $!-!$ commutative-cut. Since there is always a dereliction on the side door the box being substituted in will eventually be opened.

³This section was work done with Vincent Danos, and owes much to him.

$$\begin{aligned} !(D_x(P)) \cdot_x !(Q) &= !(D_x(P) \cdot_x !(Q)) \\ &= !(P \cdot Q) \end{aligned}$$

Which can be shown as:

$$\begin{array}{c} \boxed{\begin{array}{cc} \vdots & \vdots \\ A & ?B^\perp \end{array}} \quad \boxed{\begin{array}{cc} \vdots & \vdots \\ !B & ?C \end{array}} \\ \hline !A \quad \quad ?B^\perp \quad !B \quad \quad ?C \\ \Downarrow \\ \boxed{\begin{array}{ccccc} \vdots & & \vdots & & \vdots \\ A & & B^\perp & B & ?C \end{array}} \\ \hline !A \quad \quad \quad \quad \quad ?C \\ \vdots \quad \quad \quad \quad \quad \vdots \end{array}$$

An analysis of the three types of path that can occur in this net both before and after the cut induces certain equations. We will write e for $t!(d)$.

Type I $!(\phi)e^*(\psi)e!(\vartheta) \Rightarrow !(\phi)!(\psi)!(\vartheta)$

Type II $e!(\phi)e = ee!(\phi) \Rightarrow e!(\phi)$

Type III $e!(\phi)e^* \equiv e!(\phi)e^*$

Paths of Type I suggest the equation $e^*e = 1$, which indeed is valid in Λ^* : $!(d^*)t^*t!(d) = !(d^*)!(d) = !(1) = 1$. The Type II paths generate the equation $ee = e$, which is certainly *not* valid in Λ^* : $t!(d)t!(d) = tt!(d)!(d) = tt!(d)d = tt!(dd)$.

First, we look at how we can get a simpler data-flow for the side doors. Since now there is always a dereliction before every side door, there is always the $t!(d)$ configuration; we never have a t operation without the $!(d)$.

It would be nice if we had $t!(d) = 1$, however, the following shows that we don't:

$$\begin{aligned} t!(d)\langle a, b \rangle &= t\langle a, \langle \square, b \rangle \rangle \\ &= \langle \langle a, \square \rangle, b \rangle \end{aligned}$$

So what's the gain from TLL? In fact we can work with this equation, and there is a simple intuition to justify this rule for the token pushing implementation of the Geometry of Interaction Machine. Recall that the t operator saves a level of the environment until we return into the box on the same port. Since the environment

saved is empty, there was no need to save it, and we can safely work without the t operator. Hence, one should think of lifetime of contexts. The lifetime of the new context generated inside the box is just until it reaches the side door; the new context is not used.

We can now work in a simpler system where we do not have the t operator. The immediate consequence of this is that contexts are now *lists* rather than trees. A side door operation is now just to leave the current level of the environment. This observation is by no means restricted to TLL. We can use this as an optimisation for the general case, but only in TLL are we guaranteed that all boxes are of this form. Having shown a simpler system, we must now justify the usefulness of the language by stating what functions can be encoded into TLL.

In fact things are not so straightforward since it makes a great difference as to which encoding of the λ -calculus we use.

However, from a result in category theory we already have:

Theorem 6.4.5 In the free autonomous category with a *Natural Numbers Object* it is possible to represent all the primitive recursive functions over the natural numbers.

PROOF: See [MRA93]. □

This says that all the first order primitive recursive functions are definable in Linear Logic *without* the exponentials. Since TLL is a system with restricted exponentials we at least have all the power of a system without them. A more precise characterisation of the class of functions definable in TLL must remain an open question of this thesis.

Other local optimisations

The above presentation of TLL has shown that in a proof structure, if we have a dereliction just before a promotion, then we can, for that edge, eliminate both the d and the t operator. Since both these operations are “expensive” this is a worthwhile optimisation. This kind of optimisation is best analysed by looking at the original graph before compilation, and marking nodes that do not need code generated for them.

There are other optimisations of this nature, here we mention two.

- In the light of the above optimisation, we can analyse a special case when we have just a boxed dereliction $!^y_v(D^x_u(I_{x,y}))$. The elimination of the dereliction and the side door leave a net which is just a boxed axiom link: $!^y_v(I_{x,y})$. All paths that go through this part of the net will simply increase the level of the

environment (ep) then decrease. Again, this can be identified on the original net before compilation, and in fact combined with the previous optimisation is nothing more than η -reduction (cf. Definition 2.4.1) for the exponentials.

- Performing η -reduction in the λ -calculus also yields a compilation of less instructions for obvious reasons. The resulting code will also be slightly more efficient.

6.4.3 Global Optimisations

Perhaps the main criticism that can be put on this data flow notion of computation is the length of the path. We have looked at minor adjustments to try to reduce the length, but none of them change the order of magnitude. Here we will glean some insight into implementing paths by recalling some properties about paths. There is a property of paths, (Theorem 4.5.9) that states that they are well-balanced; they return back in a well balanced fashion. Using this information it is possible to reduce the length of the path by half at the cost of introducing an additional stack of return addresses and contexts.

Questions and answers

The property that we are going to make significant use of here is the notion that paths return back to the same place to answer the question asked, and moreover, this preserves the so-called well bracketed condition. What we shall outline here is a technique to avoid returning back along the execution path with each answer; we will jump back with the result to the previous question asked, and “restore” the token structure.

We begin with an overview, then go on to give an encoding of this idea as a modification of the Geometry of Interaction Machine.

Recall from Chapter 4 that a sequence of contexts (the trace of a path computation) gives rise to a well-bracketed sequence of questions and answers. In particular, there are critical points in the trace that ask questions of *base type*⁴, for example the boolean test for a conditional, and the argument to a function of base type (e.g. **succ**). At each question q_i asked, we will store the context, and a return address on a stack of *questions to be answered*. The computation continues (maybe asking more questions, answering questions, etc.) until we come to the point where

⁴The question and answer discipline is by no means limited to questions of base type, but if we introduce this optimisation for all nodes that the token passes through then there is no gain at all; in fact the implementation would be much worse.

we are answering the question q_i . At this point, rather than traversing the path backwards to where the question was asked, we pop the question stack, restore the current token to be the state it was when it asked the question together with the answer, then return to the place where the question was asked. The correctness of this optimisation is given by the fact that in Λ_{pcf}^* we have $\phi^*\phi = 1$ for execution paths of programs at base type, hence the contexts are equal when we ask a question and when we answer it. This then gives a optimisation on the path computation: it reduces the length of the execution path by half.

We henceforth need to modify the general structure of the Geometry of Interaction Machine to handle this feature. The only change that we need make in fact is the encodings of the constants, arithmetic functions and the conditional for PCF.

We introduce a *question stack* (qs) which we define as:

$$qs ::= (\mathcal{M}, \mathcal{E}, l) : qs \mid \square$$

where $\mathcal{M}$ and $\mathcal{E}$ are the multiplicative and exponential contexts respectively, and l is a label — the return address.

To avoid repeating the diagrams for the data flow, we refer the reader to the diagrams given for the coding of the PCF constants given in Subsection 6.3.4; in particular, we will use the same names for the labels.

Constants at base type

A path entering a constant is answering a unique question. We must examine the question stack to restore the context, update the context with the answer, and branch to the place that the question was asked. Hence we have the following code, which we will write in an informal notation:

```

l: let (M,E,l') = top qs
    let qs = pop qs
    let context = (M,E,c)
    br l'

```

Constant functions

Depending on which direction the token arrives from, we are either asking a new question, or answering a previous question.

Arriving on $l1$ is asking a new question, so the current context is pushed onto the question stack together with the return address. The token then continues its journey

as before. Arriving on $l2$ with a context (M', E', c) is answering a question, so we update the token and the question stack and branch to the place that the question was asked. This is made precise by the code fragment:

$l1:$ $\text{let } qs = \text{push } (M, E, l2) \text{ } qs$ $\text{br } l2_out$	$l2:$ $\text{let } (M, E, l) = \text{top } qs$ $\text{let } qs = \text{pop } qs$ $\text{let token} = (M, E, f \text{ } c)$ $\text{br } l$
--	--

Conditional

Arriving at the top of a conditional requires that we ask a question for the boolean argument:

$l1:$ $\text{let } qs = \text{push } (M, E, l2) \text{ } qs$ $\text{br } l2_out$
--

Arriving on each of $l2$, $l3$ or $l4$ is exactly the same as the unoptimised version.

This first substantial optimisation to the Geometry of Interaction Machine of course is not obtained for free — it is a straightforward trade of space for time.

Our implementation is still far from being very efficient simply because we have no concept of *sharing* results of computations. We will now hint at this issue in a preliminary way.

Memoisation

The concept of *memoisation* (see for example [FH88]) is to store the values computed together with their arguments, so that if we ask for the same computation to be performed again, we can immediately return the value by looking up the result in a *memo table* rather than recomputation. In particular functions like Fibonacci gain heavily from such an optimisation.

In the framework of the Geometry of Interaction Machine there is an analogous idea to this to avoid recomputation of arguments. In the graphs that we compile there is a very clear place where information can be stored about shared results: in the contraction nodes. The general idea is the following: when we enter a contraction node, we look to see if there is a value already there. If there is not, we continue as usual. If there is a value there, then we pick this value up and return back along the path that we came on. Of course, we now have to say how a value can be put into a contraction node. When we enter a contraction node from the conclusion, we are returning with a result, which we store locally to the contraction node, and continue as usual.

However things are not so simple in that value stored in the fan node must depend (be a function of) on the context. Hence we require not a single value, but an environment associating contexts to values. Hence each time we require to look up a value we must first do some sort of comparison of contexts to see which value we require. One would hope that this notion could lead to a idea of a closure being represented by a context, and then a binding would be defined as a context transformation. However, this must be left to further work.

Although not implemented as part of our implementation, we believe that combining this with the previous optimisations that we outlined gives a significant speedup of our implementation to a level where it should become very practical. Further experimental work is required to verify this claim so that it can be compared with existing implementations.

6.5 Discussion

In this chapter we have considered a novel idea for implementing a simple functional programming language based on the notion of path computations from the Geometry of Interaction. This generated a very compact coding of a simple functional language which has a remarkably simple (and small) runtime system. We propose that the basic machine could have applications where space is of the highest concern, and efficiency is not so important.

The extended system with all the optimisations that we suggested starts to look like a promising alternative to extant implementation techniques. The implementation comes directly from the underlying semantics, and all optimisations come from the underlying theory of Λ_{pcf}^* . Experience with a prototype implementation has left the author with very positive feelings about this paradigm, but we are not yet at a stage where we can produce theoretical results regarding the efficiency of this paradigm with respect to extant implementation techniques.

The compilation technique is very simple, and the ideas seem to be general enough to extend to something like Standard ML [MTH90, MT91], as we hinted. There is still a great deal of work that we need to do to achieve this goal, but the way forward seems clear; we hope to be able to report on this development in the not too distant future. Only then will we really be in a position to talk about performance issues; Standard ML seems to have become the *de facto* standard implementation to which other implementations should be compared.

It has been one of our main objectives to develop a simple stack implementation of Λ_{pcf}^* — to avoid garbage collection on the context data structure. The work on

TLL was the key to this, but we have not been able to characterise what class of functions we can contain. This remains one of the main open questions of this thesis. Finally, we remark on some additional optimisations that may be considered in future implementations of the Geometry of Interaction Machine.

- Partial evaluation of the term using the interaction net implementation given in the previous chapter. For example, reduce all the non-copying redexes. The correctness of such optimisations is guaranteed by the path invariance results of the previous chapter.
- Better representations for contexts. In particular, our implementation was based on a dynamic data structure. Our optimisation based on questions and answers in particular has to copy part of this structure which leads to substantial overheads that we are investigating reducing.
- Finally, a remark about optimal reduction. From the work of Asperti and Laneve [AL93b] there is a connection between labels and paths that we mentioned in Chapter 4. This result gives a notion of Lévy optimality for our data-flow implementation, which can be stated as roughly never asking for the same value twice. This indeed was the motivation behind the memoisation technique that was suggested. However, we must leave this to future work.

Chapter 7

Conclusions and Further Work

In this chapter we conclude and suggest further directions.

7.1 General Conclusions

Our programme of research was the investigation of the applicability of Linear Logic and the Geometry of Interaction to programming language theory; specifically implementation. In particular, this thesis has shown the Geometry of Interaction as:

- a proof technique for a Linear Logic based implementation; and
- an implementation technique itself

for a simple functional programming language (PCF). Hand-in-hand with the theoretical development of this work was the implementation of various prototypes which gave global insights and run-time performances of the ideas.

It is now time to answer the question whether Linear Logic and the Geometry of Interaction really were of use in this thesis, and if they hold sufficient content for the future.

Linear Logic

We are not alone in the belief that Linear Logic has something to say about functional language implementation. Both before the start of this thesis and during its development there has been a substantial increase in interest and results supporting this. The development of the connections with optimal reduction and reductions in Linear Logic has provided a clear understanding of how to implement this theory.

Albeit an expensive process to implement, Linear Logic at least provides a framework which we can comfortably work in. The ultimate goal of our work is to develop an implementation for optimal reduction for functional programming languages in which the cost of the β -reduction steps are no more than linear. We have tried to attack the problem from a different angle in this thesis by looking at something that is not optimal, then tried to improve it. We have not succeeded in achieving our goal, but we have gained great insight in to the problem and new ideas are being developed all the time.

Geometry of Interaction

One of the goals of this thesis is to make the ideas of the Geometry of Interaction clear by providing a simple implementation. We believe we have achieved this goal, and we feel that we now have a new perspective on the notion of reduction in the λ -calculus. We tried to show that we can make these ideas practical for simple programming languages, and indeed our prototype implementations have left us with nothing but positive feelings about this approach. Again, the end of the story is not here: there are new developments in the Geometry of Interaction itself, and new implementation techniques are following. We see that the Geometry of Interaction is without doubt an important contribution to the theory of computation, and we hope that the future for language implementation will benefit greatly from it.

With hindsight, and in the light of recent results in Game semantics, we feel that the results of this thesis would be better motivated by looking at a more general framework than paths and using Games as a motivating paradigm and proof technique.

In particular, paths are not a model of Linear Logic, so we had to take care and restrict our correctness results for the interaction net implementation net and data flow machine, much in the same way that Girard had to restrict his soundness result. Games have greater structure and in particular are a model of Linear Logic hence we could have greatly simplified and extended our results in full generality.

However, as far as the actual implementation is concerned, we hold strongly the opinion that paths are the right notion to get hold of. They are far more intuitive, and, as we have demonstrated, trivial to implement.

7.2 Interaction Nets

Our interaction net implementation was surprisingly simple, and succeeds in implementing the λ -calculus in a reasonably efficient way. As our benchmark results

show, for realistic programs where types don't grow too high, we have the machinery for a simple parallel implementation which performs better than extant optimal reducers.

We make no claim that this is in any way the “best” algorithm, or that this work is complete. We hinted at some alternatives to our algorithm that certainly deserve closer analysis.

There are a number of additional variants of our interaction net that have been investigated, but not yet completed to the level to know if they are correct, and of any use. Here we will very briefly mention two.

Telescoping boxes To overcome the problem of implementing the commutative conversion $!(P) \cdot_x !(Q) = !(P \cdot_x !(Q))$ in an interaction framework, we propose the following as a possible solution:

The basic idea is to have principal ports on the side door nodes facing towards the outside of the box (rather than point to the promotion node). It is now possible to define a sequence of interactions that will give the commutative conversion on boxes; and we leave the details of this for the reader.

This slight modification leads to a change in the kind of reduction that we get. In particular we have the following:

- Copying of a box will never complete unless the box is closed.
- As a consequence of the above, reduction works “inside out” in that we require that all substitutions are brought in from the bottom before we can continue reduction. The intuition is a *telescoping* analogy.

Whether this idea works in general is unknown since it appears that the net might deadlock for certain cyclic structures. We leave the completion of this idea to further work.

List of principal ports To overcome the problem of implementing the contraction followed by a weakening $C_z^{x,y}(W_x(P)) = P$ in an interaction net, we propose the following as a possible solution:

The basic principle is the notion of a *lazy* box, in that we delay the cut “to make sure” that the cut was really required. In particular, the cut of a contraction against a box will not copy the box, (because a weakening node might delete one copy), but we will record the fact that another copy was requested by implementing a count. A weakening will simply decrement the count, and the dereliction does the real work by making a copy of the box.

The rôles of contraction, dereliction and weakening can be formalised as follows,

Weakening Decrement copy count: $!^n A \multimap !^{n-1} A$

Contraction Increment copy count: $!^n A \multimap !^{n+1} A$

Dereliction Make copy: $!^n A \multimap A \otimes !^{n-1} A$

Where we write $!^n$ to represent that n copies have been requested so far for a particular box.

It is possible to make this idea into an interaction net, but we have not yet analysed any properties of the resulting system. As we stated, this is not a completed piece of work, and in particular, the author believes that there maybe problems in making this work in an interaction net framework. The “potential” problem is the same as above: deadlock. Since the list is ordered, we *may* be forced into a situation where the dereliction on a box cannot be performed.

Again, we leave the completion of this work to the future.

7.3 Geometry of Interaction Machine

Our work on the Geometry of Interaction machine, coding a semantics directly into assembly language, was a preliminary investigation into the possibility of using these ideas for executing real programs. The ideas are very novel, and far removed from any extant coding of functional programming languages. Our theory and practical experience with an implementation however has left the author with mixed feelings on the future of these ideas. Without doubt some of the salient features, such as small object code and small run-time system, make its use highly favourable for applications where space can be traded for time. Having a tight control over space usage on modern machines is not a serious problem, however, we envisage such applications as embedded systems, etc. where we have for the first time a possibility of a controlled space for execution of recursive programs, written in a functional language.

The story here is far from complete. Our investigations were to develop the theory and experiment with a prototype implementation.

7.4 Further work

Here we look at some more general long term goals of our research programme of using Linear Logic and the Geometry of Interaction to gain further insight into the theory and practice of programming languages.

We have seen two very different implementations of the λ -calculus coming from very natural ideas that exist in Linear Logic.

However, we have the following (quite disappointing) disadvantages:

Interaction Nets:

- No control over evaluation order. With our presentation of Interaction Nets, we cannot restrict evaluation within boxes (see Lafont [Laf92] for alternative approach).
- Because of the above point there are problems for implementing constructs like SML references—we need to ensure that the correct order of side effects is maintained.
- From an implementation view point, Interaction Nets have not been successfully implemented in an efficient way.

Data-flow:

- The Geometry of Interaction machine presented has excellent local performance, but globally we have quite a disappointing result—worst case exponential length of path.

In this section we will look at ways of overcoming some of these problems by combining ideas from both systems.

- Adding Control to Interaction Nets.
- Adding dynamics to data flow.

We will look at each of these ideas in turn.

7.4.1 Adding Control to Interaction Nets

The basic idea is to pass a token around an Interaction Net. This token is designated the task of firing a re-writing rule in a net. We therefore need a slightly modified definition of an Interaction Net — as before, but with the additional restriction that a rule can fire only if the rewrite is enabled. A rewrite is enabled iff both nodes involved in the interaction are enabled.

We now need to state how to enable nodes. Consider an Interaction Net as before, with an initial labelling function setting all nodes in the network to inactive. Evaluation proceeds by pushing a token around the network (*cf.* Chapter 6).

1. A token passes freely along axiom links.
2. When a token passes through a node, the node becomes enabled and we proceed as dictated by the token.
3. When the token is on a cut link (this is identified by the token) the token is pushed back through the previous node, leaving the cut edge free for interaction, which is now possible since both nodes connected by the cut link are enabled.

This method of reduction continues until the token emerges at the root of the network, at which time the net is in head normal form, and all reductions performed were outermost.

The correctness of this algorithm comes directly from results obtained in the previous two pieces of work.

There are other ways to add control to a network which will not be discussed here.

7.4.2 Adding Dynamics to Data flow

An almost dual idea to the last implementation can be obtained by adding dynamics to a (previously) static network. The intention is that we do not want to re-compute paths that have already been traversed. This is the general theme behind the work of Danos and Regnier on virtual reduction [DR93]. One can regard their work as trying to reduce the length of the path so that it is linear in the number of cut-elimination steps, rather than exponential.

7.4.3 Program Analyses

Our programme of research in general terms is looking at how to devise new implementation techniques, frameworks for representing computation, and optimising these approaches.

In this thesis we have not spoken very much about the optimisation and analysis phase of this work, and it is this that will be our next step. In particular, one general long term goal for this work is to investigate the use of Linear Logic and the Geometry of Interaction for Static Analysis (Abstract Interpretation).

Linear Logic is about using data zero, once, or more than once. This kind of information is coded into the types of programs. Knowing that an argument to a function is used exactly once has major applications in compiler technology for the implementation of safe side effects and garbage collection issues. Hence, Linear Logic should have something to say about this issue. Recent work on usage types, operational linearity, etc.

Recent work of Wadler [WT94], (and others), have indicated that there is hope in this. What we propose is the use of this semantic framework, the Geometry of Interaction as a basis for computing such properties as strictness.

In particular, we highlight two pieces of work

1. Can we use these techniques to extract existing properties about programs
2. Can we extract new properties.

The infancy of this work prohibits us from being more precise at this moment.

7.5 Summary

With respect to the introduction, we must now ask if we have achieved our initial goals. We believe that the work of this thesis has provided new ways at looking at implementation of functional programming languages.

We have devised a new way of implementing reduction of programs in an interaction net framework that we claim is more efficient than optimal reducers simply because of the reduced bookkeeping. However, we have not achieved our ultimate goal of implementing optimal reduction without additional bookkeeping. Never-the-less, we feel that this step is a contribution to the understanding of the problem.

We have devised a new way of looking at Girard's Geometry of Interaction interpretation of for Linear Logic. The simplicity of this interpretation has provided a clear, computational way of understanding the notion of a path computation. The implementation itself leads to a new way of thinking about computation, and we believe that this work can provide a solid basis on which we can build more realistic implementations so that we are in a position to judge the relative merits. It is too soon to know if this implementation technique really will be more efficient. However experiments with an implementation (and also execution on paper) have left the author with strong, positive, conclusions. We hope the report on these results in the near future.

Appendix A

GOI-Tools

In this appendix we give a brief overview of a piece of software developed in parallel with this thesis to test out our ideas and obtain “global” intuitions and benchmark results.

A.1 Introduction

GOI-Tools is a prototype tool developed in parallel with the research done in this thesis to test out the ideas, obtain benchmark results and obtain deeper intuitions about the ideas presented. In this appendix we will mention some of the features and details of the implementation, together with information and examples of how to use the system.

The implementation is written in Standard ML and C and provides the following main features:

- The interaction net algorithm presented in Chapter 5.
- The Geometry of Interaction Machine presented in Chapter 6.

In addition, GOI-Tools provides many useful reports on a computation, which are generated in L^AT_EX:

- Traces of evaluation on labelled λ -terms. This tool was used for many examples in this thesis.
- Trace of the Geometry of Interaction Machine showing the sequence of transformed contexts, for example that produced in Figure 6.2.

Girard’s Geometry of Interaction is a low level decomposition of computation, where *local* steps of reduction are very simple, atomic, operations. Having an implementation

available has allowed us to get more of a *global* understanding of this notion of computation, and in particular observing the traces of the Geometry of Interaction Machine has lead the author to an understanding of the connection between paths and Games which we hinted at in Chapter 4, and has lead to global optimisations for the Geometry of Interaction Machine given in Chapter 6.

A.2 Interaction Nets

There are several implementations of general Interaction Nets, for example those reported in Lafont [Laf89] and Gay [Gay91]. Asperti and Laneve have produced an implementation of Gonthier et al's work. which is a specific interaction net for implementing the optimal reduction algorithm.

Our implementation is also based on a specific interaction net as given in Chapter 5 for PCF, extended with arithmetic operators (Subsection 5.5.1).

There seems to be a general understanding of how to implement interaction nets sequentially, but unfortunately there is to date very little published on this subject. To our knowledge, the only documented account is given in [Gay91], which we recommend very strongly.

The implementation is written totally in Standard ML. A graph representation is built for the term, and the interaction net rules of Chapter 5 are coded as simple re-write rules on this data-structure.

A.3 GOI Machine

The implementation of the Geometry of Interaction Machine has already been discussed in detail in Chapter 6. Using the graph data-structure for the term used for the interaction net implementation as a starting point, we produce an output in C as specified in Chapter 6. This is then linked with some standard functions to represent the context data-structure and then compiled through a C compiler resulting in a stand-alone executable object.

A.4 Using the system

The language that we have implemented is built up from the Linear $\lambda\sigma$ -calculus together with the PCF constants that we have studied, and extended with some simple arithmetical constants.

```

program ::= let ID = EXP
           | EXP
EXP ::= fn ID ⇒ EXP
        | if BEXP then EXP else EXP
        | [ID = <ID,ID>] EXP
        | [ID = _] EXP
        | EXPA
        | (EXP EXP)
        | (EXP)
EXPA ::= EXP + EXP
        | EXP - EXP
        | EXP * EXP
        | EXP div EXP
        | EXPB
EXPB ::= succ EXP
        | pred EXP
        | iszero EXP
        | Y EXP
BEXP ::= EXP = EXP

```

Figure A.1: Syntax for GOI-Tools

The syntax of the language accepted by GOI-Tools is given in Figure A.1.

The system is invoked by executing the SML executable for the unix prompt:

```
% goi-tools
```

```
val it = true : bool
GOI-Tools Version 1.4
```

```
->
```

where `->` is the GOI-Tools prompt.

The following commands are available, which are shown on-line by the `/h` command:

```
/q quit the system
/c toggle compiler mode
/d debug mode
/x Latex output
/h help information
/n !D -o D mode (not yet implemented)
/v !(D -o D) mode
```

- Toggle compiler mode: Used to toggle between the interaction net evaluator and the GOI Machine compiler.
- Debug mode: Used to switch on the tracing for the GOI Machine. (This will only have effect if GOI-Tools is in compiler mode). This is used to produce a latex trace of the computation, i.e. as shown in Example 6.3.1.
- Latex output: Used to produce a sequence of snapshots during reduction of the labelled PCF terms.

`/q` and `/h` have obvious meanings, and the different encoding of PCF into Linear Logic proof structures is not currently available; only the $!(D - oD)$ translation is available.

All other things that are typed at the `->` prompt will be evaluated. We give a simple annotated session:

```
->let i = fn x => x ;
<function>
Number of Interactions:0
```

Number of Beta reductions:0

Number of weakenings:0

->

Here we have defined the identity function, and bound it to the identifier *i*. The function was evaluated, and the number of interactions are printed, together with information stating how many of the interactions were β -reductions and how many were weakenings (garbage collections). In this case, since the program was in normal form, there were no reductions.

We can now apply this function to an argument:

->i 4 ;

4

Number of Interactions:1

Number of Beta reductions:1

Number of weakenings:0

->

Here are some more examples:

->let k = fn x => fn y => [y=_] x ;

<function>

Number of Interactions:0

Number of Beta reductions:0

Number of weakenings:0

->let s = fn x => fn y => fn z => [z=<u,v>](x u)(y v) ;

<function>

Number of Interactions:0

Number of Beta reductions:0

Number of weakenings:0

->s k i tt ;

tt

Number of Interactions:8

Number of Beta reductions:6

Number of weakenings:1

->

Finally we show the last example when run through the GOI Machine:

Compiler mode set

->s k i t t ;

->

From the unix prompt we can execute the resulting code:

% out1

Finished after 103 stack operations

Result :tt

A.5 Availability

GOI-Tools is available by anonymous ftp from:

theory.doc.ic.ac.uk:theory/software/GOI-Tools.

The files are:

- goi-tools: The SML executable
- header.c, footer.c: library functions used by the GOI Machine
- goi-tools.dvi: documentation

It is assumed that the user has the gcc compiler. If you require the system to work with a different compiler, please contact the author.

Bibliography

- [Abr90] Samson Abramsky. The lazy λ -calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, chapter 4, pages 65–117. Addison Wesley, 1990.
- [Abr91] Samson Abramsky. Proofs as processes. Unpublished Lecture Notes, 1991.
- [Abr93] Samson Abramsky. Computational Interpretations of Linear Logic. *Journal of Theoretical Computer Science*, 111:3–57, 1993. Earlier version appeared as Imperial College Technical Report DOC 90/20, October 1990.
- [ACCL91] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.
- [AD94] Andrea Asperti and Giovanna Dore. $\delta o! \varepsilon = 1$, February 1994.
- [ADLR94] Andrea Asperti, Vincent Danos, Cosimo Laneve, and Laurent Regnier. Paths in the λ -calculus: Three years of communication without understanding. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 426–436. IEEE Computer Society Press, 1994.
- [AH87] Samson Abramsky and Chris L. Hankin, editors. *Abstract Interpretation for Declarative Languages*. Ellis Horwood, 1987.
- [AJ94] Samson Abramsky and Radha Jagadeesan. New foundations for the geometry of interaction. *Information and Computation*, 111(1):53–119, 1994.
- [AJM94] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF (extended abstract). In Masami Hagiya and John C. Mitchel, editors, *Theoretical Aspects of Computer Software. International*

- Symposium TACS'94*, number 789 in Lecture Notes in Computer Science, pages 1–15, Sendai, Japan, April 1994. Springer-Verlag.
- [AK93] Zena M. Ariola and Jan Willem Klop. Cyclic lambda graph rewriting. Slides from Semagraph II workshop, Nijmegen, June 1993.
- [AL91] Andrea Asperti and Cosimo Laneve. Interaction systems I: The theory of optimal reductions. Technical Report 1748, INRIA Rocquencourt, 1991.
- [AL93a] Andrea Asperti and Cosimo Laneve. Interaction systems II: The practice of optimal reductions. Technical Report UBLCS-93-12, Laboratory for Computer Science, University of Bologna, May 1993.
- [AL93b] Andrea Asperti and Cosimo Laneve. Paths, computations and labels in the λ -calculus. In Claude Kirchner, editor, *Rewriting Techniques and Applications. 5th International Conference, RTA-93, Montreal, Canada.*, number 690 in LNCS, pages 152–167. Springer-Verlag, June 1993.
- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [Bar84] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.
- [Baw86] Alan Bawden. Connection graphs. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 258–265, 1986.
- [BB90] Gérard Berry and Gérard Boudol. The Chemical Abstract Machine. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 81–94, 1990.
- [Bec92] Denis Bechet. Partial evaluation of interaction nets. *Journées Francophones des Langages Applicatifs*, February 1992.
- [Big89] Norman L. Biggs. *Discrete Mathematics*. Oxford Science Publications. Clarendon Press, revised edition, 1989.
- [Dan90] Vincent Danos. *Une Application de la Logique Linéaire à l'Étude des Processus de Normalisation (principalement du λ -calcul)*. PhD thesis, Université Paris VII, June 1990.
- [deB72] N. G. deBruijn. Lambda calculus notation with nameless dummies. *Indagationes Mathematicae*, 34:381–392, 1972.
- [DR93] Vincent Danos and Laurent Regnier. Local and asynchronous beta-reduction (an analysis of Girard's execution formula). In *Proceedings, Eighth Annual*

- IEEE Symposium on Logic in Computer Science*, pages 296–306. IEEE Computer Society Press, 1993.
- [DR94] Vincent Danos and Laurent Regnier. Proof-nets and the Hilbert space, 1994. Submitted for publication.
- [FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison Wesley, 1988.
- [GAL92] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *Proceedings of ACM Symposium Principles of Programming Languages*, pages 15–26, January 1992.
- [Gay91] Simon J. Gay. Interaction nets. Diploma in Computer Science Dissertation, University of Cambridge Computer Laboratory, 1991.
- [Gir71] Jean-Yves Girard. Une Extension de l’Interprétation de Gödel à l’Analyse, et son Application à l’Elimination des Coupures dans l’Analyse et la Théorie des Types. In Jens Erik Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63–92, Amsterdam, 1971. North-Holland.
- [Gir87] Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
- [Gir88] Jean-Yves Girard. Geometry of interaction 2: Deadlock-free algorithms. In P. Martin-Löf and G. Mints, editors, *International Conference on Computer Logic, COLOG 88*, pages 76–93. Springer-Verlag, 1988. Lecture Notes in Computer Science 417.
- [Gir89a] Jean-Yves Girard. Geometry of interaction 1: Interpretation of System F. In R. Ferro et al., editor, *Logic Colloquium 88*. North Holland, 1989.
- [Gir89b] Jean-Yves Girard. Towards a geometry of interaction. In J. W. Gray and Andre Scedrov, editors, *Categories in Computer Science and Logic*, volume 92 of *Contemporary Mathematics*, pages 69–108. American Mathematical Society, 1989.
- [Gir94] Jean-Yves Girard. Geometry of interaction III : accommodating the additives, 1994. Laboratoires de Mathématiques Discrètes, University of Marseille.

- [GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
- [Han94] Chris Hankin. *λ -calculi*. Graduate texts in Computer Science. Oxford University Press, September 1994.
- [HO93] J. Martin E. Hyland and Luke Ong. An approach to (intensional) full abstraction for PCF, July 1993. Preliminary announcement.
- [Hol88] Sören Holmström. Linear functional programming. In T. Johnsson, Simon L. Peyton Jones, and K. Karlsson, editors, *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, pages 13–32, 1988.
- [How80] W.A. Howard. The formulae-as-types notion of construction. In J.P. Hindley and J.R. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.
- [Hue90] Gérard Huet, editor. *Logical Foundations of Functional Programming*. Addison Wesley, 1990.
- [Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, 1974.
- [Kat90] Vinod K. Kathail. *Optimal Interpreters for Lambda-calculus Based Functional Programming Languages*. PhD thesis, MIT, May 1990.
- [Klo80] Jan Willem Klop. *Combinatory Reduction Systems*, volume 127 of *Mathematical Centre Tracts*. Mathematischen Centrum, 413 Kruislaan, Amsterdam, 1980.
- [KR88] Brian W. Kernighan and Denis M. Ritchie. *The C Programming Language*. Software Series. Prentice Hall, second edition, 1988.
- [Laf87] Yves Lafont. *Logiques, Catégories and Machines*. PhD thesis, Université de Paris 7, September 1987.
- [Laf89] Yves Lafont. The sketchy manual of Interaction Nets (v0), December 1989. Laboratoire d'Informatique de l'Ecole Normale Supérieure.
- [Laf90] Yves Lafont. Interaction nets. In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 95–108. ACM, ACM Press, January 1990.

- [Laf92] Yves Lafont. The paradigm of interaction. In *Proceedings of the workshop on Concurrency*, pages 276–293. Programming Methodology Group, Göteborg, April 1992.
- [Laf94] Yves Lafont. From proof-nets to interaction nets, March 1994. Laboratoires de Mathématiques Discrètes, University of Marseille.
- [Lam90] John Lamping. An algorithm for optimal lambda-calculus reductions. In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 16–30. ACM, ACM Press, January 1990.
- [Lan64] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.
- [Lan93] Cosimo Laneve. *Optimality and Concurrency in Interaction Systems*. PhD thesis, Dipartimento di Informatica, Università degli Studi di Pisa, March 1993.
- [Lev78] Jean-Jacques Levy. *Réductions Correctes et Optimales dans le Lambda-Calcul*. PhD thesis, Université Paris VII, January 1978.
- [Lev80] Jean-Jacques Levy. Optimal reductions in the lambda calculus. In J.P. Hindley and J.R. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, 1980.
- [Mac91] Ian Mackie. Lilac : A functional programming language based on linear logic. Master’s thesis, Department of Computing, Imperial College of Science, Technology and Medicine, September 1991.
- [Mac94] Ian Mackie. Lilac : A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):1–39, October 1994.
- [Mil90] Robin Milner. Functions as processes. In *Proceedings of ICALP 90*, volume 443 of *Lecture Notes in Computer Science*, pages 167–180. Springer-Verlag, 1990.
- [Mos79] Peter Mosses. SIS — semantics implementation system, reference manual and user guide. DAIMI Report MD-30, DAIMI, University of Århus, Denmark, 1979.
- [MR91] Pasquale Malacaria and Laurent Regnier. Some results on the interpretation of λ -calculus in operator algebras. In *Proceedings, Sixth Annual IEEE*

- Symposium on Logic in Computer Science*, pages 63–72. IEEE Computer Society Press, 1991.
- [MRA93] Ian Mackie, Leopoldo Román, and Samson Abramsky. An internal language for autonomous categories. *Journal of Applied Categorical Structures*, 1(3):311–343, 1993.
- [MT91] Robin Milner and M. Tofte. *Commentary on Standard ML*. MIT Press, 1991.
- [MTH90] Robin Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International, 1987.
- [Plo77] Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
- [Reg92] Laurent Regnier. *Lambda-Calcul et Réseaux*. PhD thesis, Université Paris VII, January 1992.
- [Rey74] John C. Reynolds. Towards a theory of type structure. In *Paris colloquium on programming*, volume 19 of *LNCS*. Springer-Verlag, 1974.
- [Rov92] Luca Roversi. A compiler from curry-typed- λ -terms to the linear- λ -terms. In proceedings of the IV Convegno italiano di informatica teorica, World Scientific Publishing Company, 1992.
- [Sch94] Harold A. Schellinx. *A Hitchhiker's Guide to Linear Decorating*. PhD thesis, Institute for Logic, Language and Computation. University of Amsterdam, 1994.
- [Tur91] Raymond Turner. *Constructive foundations for Functional Programming*. McGraw Hill, 1991.
- [Wad71] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, Oxford University, 1971.
- [Wad90] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee,

Israel, April 1990. IFIP TC2 Working Conference on Programming Concepts and Methods, North-Holland.

[Wak90] David Wakeling. *Linearity and Laziness*. PhD thesis, University of York, 1990.

[WT94] Philip Wadler and David Turner. Once upon a type, August 1994.