University of London
Imperial College London
Department of Computing

Curry-Howard Calculi from Classical Logical Connectives

A Generic Tool for Higher-Order Term Graph Rewriting

Jayshan Raghunandan

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of the University of London and the Diploma of Imperial College, October 2008

Abstract

This thesis contains a study of Curry-Howard correspondences for Sequent Calculus formulations of Classical Logic. Starting from Gentzen's formal definition of logical consequence (presented in the framework of his sequent calculus), we present an automated process to conservatively extend the framework with primitive logical connectives and corresponding cut-elimination rules. The key difficulty lies in deriving the main cut-elimination rules for the connective. In contrast to existing works, which employ brute-force techniques or methods using equivalences, we specify an algorithm which systematically constructs the main cut-elimination rule by operating on rows of truth-tables. (We also give a geometrical interpretation of the cut rule.) The aim is to study the computational content of the resulting formulation of classical logic.

We mechanically extract from the framework, a computational term calculus inspired by the $\mathcal{X}$-calculus of van Bakel, Lengrand and Lescanne. We motivate our design choices by making comparisons with existing computational calculi that hold Curry-Howard correspondences with Classical Logic. Using our process, we then build and study a number of computational calculi, focusing on their simulation properties. We find that notions of logical expressibility (the ability of a connective to logically express another) and computational expressibility (the ability of a term calculus to simulate another) do not coincide.

Our (graphical and interactive) tool is a full implementation of the process we design in this thesis, but additionally serves as tool for higher-order conditional term graph rewriting in general; it also features Visser's language of strategy combinators, allowing one to easily study complex reduction behaviour. We detail some specific implementation problems we encountered, and motivate the solutions we adopted.

Acknowledgements

First and foremost, I wish to thank my supervisor Steffen van Bakel for his support and encouragement over the past years. I am grateful for the many hours spent explaining the subtle and technical details of our field, and especially for the guidance and feedback given whenever required.

I would like to thank Alexander Summers, who has been a most excellent friend and colleague. The very many discussions we had on the $\mathcal{X}$-calculus (and related subjects) was undoubtedly the key ingredient that kept me motivated during the entire programme.

I am fortunate to have been located in such a 'cultured' office. I would like to thank Billiejoe, Jaspreet, Simon, Ioana and Hywel (and also Dorian) for being great friends and always making good office banter.

Finally, I would like to thank my parents for providing me with rent free accomodation and for their patience during the entire programme.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Developments in theory and practice are like reductions in a confluent rewrite system with critical pairs: they do not ride side-by-side, but every so often they have a chance to join. Notions of computability were first formalised by mathematicians and logicians long before 'computing machines' were conceived in hardware. For a while, the theory of computation led practical developments, and conceptual devices such as Turing machines were realised only on pen and paper.

Turing and Church each proposed quite different formalisms that could 'compute': Turing proposed the now infamous 'Turing machines', while Church proposed the (equally infamous) λ-calculus. Both models of computation, which were later shown to be equivalent to one-another, formalised the idea of an algorithm. Church's model was stateless while Turing's, stateful, but due to the ease in which Turing's model could be realised in hardware, the first real 'computing machines' followed Turing's design.

Since that first move, developments in practical computing accelerated and the accompanying theory struggled to keep up. Fundamental developments in computing (e.g., the introduction of high-level programming languages) were motivated by practical concerns, and were often designed by engineers who were neither logicians, mathematicians nor (more broadly speaking) theoreticians. It was this lack of a theoretical foundation that, in our opinion, led to programming language failures such as the '`GOTO`' statement, and more generally, poor levels of abstraction (e.g., the `WRITE OUTPUT TAPE` command of FORTRAN). However, noting this, we should also mention that had the field of practical computing waited for theoreticians to design great programming languages, advances would have undoubtedly been *slow*.

The gap between the two fields is not so great anymore, and it is widely accepted now that any programming language worth investing time in should have a sound theoretical foundation. Even in the case of pure hardware design, methods of *formal* verification are desired to ensure components which will be massed produced 'work', and in the catastrophic situations where they do not, to fend off lawsuits and try and relocate blame.

Church's original goals were actually related to the formalisation of mathematics, and being a logician himself, it is perhaps not entirely remarkable that an intimate correspondence between his model of computation and a logic was discovered. The discovery, now known as the *Curry-Howard Isomorphism*, was made on independent occasions by Curry and Howard, and related the type of a λ-calculus program with a formula denoting a proposition of intuitionistic logic. The correspondence also related typeable programs to logical proofs and the execution of a program to a notion of proof normalisation.

While real world programming languages and computing machines were being developed by implementors, masses of theoretical computers scientists were at work: abstracting, formalising and proving. What they found was quite remarkable: the directions taken by the implementors were not at all orthogonal to that which would have (likely) been taken by theoreticians. In fact, we can now see that the programming language features introduced by implementors had foundations that were deeply rooted in logic. Perhaps the most prominent example is the discovery of the *continuation*, which represents the dual notion of a function (it is an object that consumes, rather than produces, a result).

Griffin was the first to relate continuations with Classical Logic. He was able to type certain control operators (i.e., special functions which manipulated continuations) with formulas corresponding to propositions of Classical Logic. Before his discovery, it was folklore that Classical Logic did not have a computational counterpart since it was not constructive. After this however, a number of researchers began what is now referred to as a 'quest' to find an exact correspondence between Classical Logic and some model of computation, i.e., to find a 'Curry-Howard Isomorphism' for Classical Logic.

Progress has been steady since Griffin's discovery, but the difficulty of extracting computational content from a classical logic is in finding a suitable presentation that can also be used as a computational model. Logicians accept that the Sequent Calculus gives the best presentation of Classical Logic since it preserves all of its symmetries. However, the non-confluent cut-elimination and the permutability of proofs present problems when attempting to assign computational meaning.

A major contribution was made by Parigot, who argued that neither the Natural Deduction Calculus nor the Sequent Calculus were suitable for studying the computational content of Classical Logic. Subsequently, he introduced a logical calculus which he called 'Free Deduction', for which he defined a confluent set of rules for proof normalisation. Parigot then extracted a computational calculus called the $\lambda\mu$-calculus, which inspired a wealth of research. As a result, some of the 'classical' features of computing were made clear.

Another breakthrough was made by Herbelin, who managed to assign computational meaning to sequent calculus proofs. With the help of Curien, he later designed a computational calculus (called $\overline{\lambda}\mu\tilde{\mu}$) held a Curry-Howard correspondence with a restricted formulation of a sequent calculus for classical logic. Notably, their calculus was not confluent, yet still served well as a model of computation. A subsystem of $\overline{\lambda}\mu\tilde{\mu}$ was later identified that held an exact correspondence with a refinement of Gentzen's Sequent Calculus. This subsystem was studied by van Bakel, Lengrand and Lescanne and formed the foundation of the $\mathcal{X}$-calculus.

The $\mathcal{X}$-calculus is a computational calculus which holds the closest Curry-Howard style correspondence with Gentzen's original presentation of Classical Logic. In fact, the reduction system of the $\mathcal{X}$-calculus is also highly non-confluent, and features unjoinable critical pairs that mirror the standard cut-elimination. We are certainly not advocating that these properties as 'desirable' when designing models of computation, but an (appropriate) answer to the following question justifies their presence in $\mathcal{X}$.

"When should one restrict a Curry-Howard calculus to study computational behaviour?"

The main lines of work starting from Parigot introduced restrictions at the level of the logic, and then sought to extract a computational model. The philosophy of the $\mathcal{X}$-calculus, in contrast, seeks to extract a term calculus *directly* from the logic, and then looks at placing restrictions to gain a desirable model of computation. This latter approach has the advantage that the symmetries of the logic are preserved in the computational model. In addition, comparisons between different subsystems can be carried out within the same framework. In fact, two simple restrictions on the reduction relation have been shown to yield two confluent reduction subsystems that correspond to dual notions of computation called 'call-by-name' and 'call-by-value'.

The work carried out in this thesis has two separate concerns, but both can be related to the $\mathcal{X}$-calculus. First we are interested in understanding its complex reduction system; we do this mainly through implementation. Second, we investigate generalisations that can be made to $\mathcal{X}$, and study closely calculi built to hold Curry-Howard correspondences.

1.1 Contributions

The contributions of this thesis are listed below.

1. We provide a novel framework for specifying higher-order conditional term graph rewrite systems. The framework also features a rich language for describing reduction strategies. We provide an implementation of this framework as an open-source tool written in Java. The tool is highly modular, has a graphical component (enabling interactive reductions) and is intended to be extended with user-defined representations of term graphs. Our tool can be downloaded at:

   `http://www.doc.ic.ac.uk/~jayshan/GRT`

2. Contrary to intuition, we show that 'logical expressibility' does not imply 'computational expressibility'. That is, we show that a computational term calculus built from a set of logical connectives: (i) may not be able to fully simulate calculi built from the connectives it can logically express, and (ii) may be able to simulate calculi built from connectives it cannot logically express.

3. We specify an algorithm which can mechanically build a Curry-Howard 'pair' of calculi (a classical sequent calculus and a term calculus in the style of $\mathcal{X}$). Our contribution describes an intelligent algorithm (i.e., not employing brute-force techniques), which operates on a truth function to define cut-elimination rules. To the best of our knowledge, we are also the first to relate the main cut-elimination rule for a connective with geometrical features.

The above contributions have been consolidated in the implementation of our tool. Given a truth table, the tool will (conservatively) extend a basic sequent calculus and a basic term calculus with appropriate inference rules and cut-elimination rules and corresponding term constructors and reduction rules.

1.2 Statement of Originality

I declare that this thesis was composed by myself and that the work it presents is my own except where stated otherwise. With the exception of the following publications, and to the best of my knowledge, it contains no materials previously published or written by another person except where due acknowledgment is made in the thesis itself.

- S. van Bakel and J. Raghunandan. *Implementing $\mathcal{X}$ [10].*
- J. Raghunandan and A. Summers. *On the Computational Representation of Classical Logical Connectives [74].*

1.3 Thesis Outline

In Chapter 2, we review the works relating to the development and understanding of the $\mathcal{X}$-calculus. We detail Gentzen's Sequent Calculus presentation of Classical Logic and study several models of computation which were designed to hold correspondences with logics. We also review some of the techniques we considered when implementing our rewriting tool.

In Chapter 3, we introduce the $\mathcal{X}$-calculus itself, giving a full description of its novel syntax and complicated reduction mechanism. We compare the calculus with other computational calculi with logical foundations and present some of the optimisations we introduced to the calculus.

In Chapter 4, we give the specification for our higher-order conditional term graph rewrite system, of which our tool was an implementation. We specify the $\mathcal{X}$-calculus as an instance of a CTGRS, and study solutions we proposed to some implementation problems we encountered, in particular the problems of name capture and name clash. We extend the CTGRS formalism with a rich language for describing reduction strategies in general, and study some appropriate strategies for reductions in the $\mathcal{X}$-calculus. We quantitatively compare our proposed solutions (to name capture and name clash) through a suite of benchmarks.

In Chapter 5, we study the relationship between the type system for the $\mathcal{X}$-calculus and the logic on which it is built. We detail a 'recipe' for building Curry-Howard 'pairs' of calculi from a set of logical connectives following the style of $\mathcal{X}$. We study and relate a number of such calculi employing different sets of

logical connectives as primitive, including a calculus based on the $\mathcal{X}^{\leftrightarrow}$ connective. We find that our 'recipe' for building Curry-Howard calculi does not always build the simplest reduction rules for the connectives in question and decide this warrants further investigation.

In Chapter 6, we study the relationship between the classical truth functions of a connective and its sequent calculus style inference rules, as studied by Call. We formalise his work, specifying an algorithm that builds inference rules from truth tables. From the insights gained, we construct a reverse process which relates the inference rules for a connective back to the truth table rows for that connective. We formulate a notion of 'cut' that acts on rows of truth tables, then design an algorithm for building the 'simplest' principal reduction rules, which 'fixes' our recipe for building Curry-Howard 'pairs' of calculi.

Chapter 2

Background

This background chapter is concerned with three topics: structural proof theory, computability theory and higher order rewriting. We relate each of these topics to the $\mathcal{X}$-calculus of van Bakel, Lengrand and Lescanne [9], which is the main subject matter of this thesis.

Structural proof theory and computability theory were related by Curry [33], Howard [49] and de Bruijn [22], who independently discovered that these independent fields of scientific research were fundamentally linked together. It was discovered that provable formulas of minimal implicative logic in Natural Deduction calculus could be represented as types for the terms of the λ-calculus. Various authors have sought to extend this relationship to hold between the sequent calculus for classical logic and some model of computation. The $\mathcal{X}$-calculus is one calculus for which the correspondence does hold.

In order to study the reduction mechanism of the $\mathcal{X}$-calculus and extensions of the $\mathcal{X}$-calculus in more detail, an implementation was sought. The $\mathcal{X}$-calculus is a higher order rewriting language featuring non-standard binding structures. We will review some of the existing implementation techniques for higher order languages and comment on their suitability for an implementation of the $\mathcal{X}$-calculus.

2.1 Notation

This section describes the notation we use throughout the course of this thesis.

We will use three standard structures throughout: sets, lists and tuples. We give details on our chosen notation below.

Definition 2.1.1 (Sets) *We may construct sets using standard set comprehension notation* $\{x \in V \mid \Theta(x)\}$ *denoting the set of all elements* x *in some well defined set* V *satisfying the predicate* $\Theta(x)$*; we will omit* V *if it is clear from the context. We use the symbols* $\cup, \cap, \setminus$ *to describe the usual set operations union, intersection and exclusion. The size of a set* S *is a count of its elements and is denoted* $|S|$*.*

Definition 2.1.2 (Lists) *A list of* n *elements of a particular type is written* $[a_0, \ldots, a_{n-1}]$*. We will use the symbols ':' and '*$+\!+$*' to denote the usual list operators cons and concatenation. Direct access to an element* i *of a list* L*, indexed from zero, is permitted using standard array notation* $L[i]$*. We will write the empty list as* $[\,]$*.*

If an element a *is a member of a list* L*, we write* $a \in L$*. We also write* $L \setminus e$ *to denote a list which has* all *occurrences of the element* e *removed from it.*

Like set comprehensions, we will allow for a similar kind of 'list comprehension'. We will write $[x \in V \mid \Theta(x)]$ *denoting the list (without duplicates) of all elements* x *satisfying the predicate* $\Theta(x)$*, for some deterministic enumeration of the elements in* V *(if a deterministic enumeration is not obvious, one will be specified).*

The size of a list L *is a count of its elements and is denoted* $|L|$*.*

Definition 2.1.3 (Tuples) *If* $T = \langle X_1, \ldots, X_n \rangle$ *is an* n*-tuple, we may access the* i^{th} *element of the tuple using a projection operator, written* T_i *(with* $0 < i \leq n$*. Alternatively, we may use the more descriptive notation* T_{X_i} *when convenient.*

The Cartesian product of sets $S_1 \times \ldots \times S_m$ *is denoted by* $\prod(S_1, \ldots, S_m)$*.*

Where variables (over some well defined set) are concerned, we will use the symbol _ to represent an anonymous variable.

Our algorithms are specified in functional pseudo-code and make use of pattern matching constructs and Haskell's guard notation ('|' and '`otherwise`') when needed.

The set of natural numbers is written $\mathbb{N}$. We will sometimes express natural numbers in base-2 notation, where the most significant bit is the left-most. e.g. the natural number 5 may be written as 0101_2. If $\mathcal{T}$ is the set $\{0, 1\}$, then the function $\ddot{\overline{r}}_n :: \mathbb{N} \to [\mathcal{T}]$ translates the base-10 representation natural number r to a base-2 number that is expressed as a list with n elements according to the above conventions.

Part of this thesis is concerned with conditional term rewrite systems. We will use the notation:

$$L \rightarrow R \leftarrow S$$

To denote a rewrite rule, where L is the left hand side, R is the right-hand side and S (if present) is the side-condition of the rule.

2.2 Structural Proof Theory

In formalising a mathematical theory, one abstracts away from its meaning, leaving only its form. If at all possible, this process is non-trivial and involves embodying all properties of the theory as explicit axioms, propositions and theorems. Once formalised, one may make deductions about the theory, treating technical terms simply as words without any meaning. Kleene rationalises this: "For to say that ['words' in the formal system] have meanings necessary to the deduction of the theorems, other than what they derive from the axioms which govern them, amounts to saying that not all of their properties which matter for the deductions have been expressed by the axioms.".

Logicians are concerned with proving the validity of statements within a system of reasoning, with respect to some (interesting) metatheory. The first 'formal' logicians were concerned with building a logical foundation for mathematics. Hilbert conjectured that all of mathematics could be formalised from some finite choice of axioms which were provably consistent. Although Gödel undermined this trail of thinking with his Incompleteness Theorem, the formalisation of logical reasoning systems was an important invention in proof theory.

Several formal systems of reasoning have been built that embody the different systems of inferences used in mathematics. There are many accepted sets of inferences or *formal logics*. Intuitionistic Logic, for example, permits only constructive arguments, while Classical Logic additionally accepts indirect proofs. Mathematicians can extend these formal systems with axioms and forms of inferences for their object language of interest, then make inferences about their theory with respect to a particular logic within the corresponding system.

Hilbert's formalism for proving the validity of propositional statements consisted of a collection of axiom schemas plus *modus ponens* (see the rule $(\rightarrow E)$ below). In [41], Gentzen proposed two alternative formalisms: Natural Deduction and the Sequent Calculus; we will discuss these formalisms next.

Gentzen's natural deduction calculi were built up from a single axiom schema (stating that under some assumption, the result stated by that assumption holds) and for each sentence constructor in the propositional language, a connective in the formal language defined as a collection of inference rules describing (i) the grounds necessary to assert a proposition exhibited by the connective (its introduction), and (ii) the conclusions that could be drawn from an assertion exhibiting the connective (its elimination). For example, the rules for implication are:

$$\frac{\begin{array}{c}(A_1)\\ \vdots \\ A_2\end{array}}{A_1 \to A_2}\,(\to I) \qquad \frac{A_1 \to A_2 \quad A_1}{A_2}\,(\to E)$$

Gentzen's Natural Deduction proofs are presented as trees of statements in which each node is justified by an inference rule, and each leaf (represented with brackets) representing a basic axiom has been discharged. This configuration was intended to mimic the style of reasoning that mathematicians followed, and thus gained popularity in preference to the Sequent Calculus and Hilbert-style systems. A major drawback, however, was that constructing a proof relied heavily on ones natural ability to build arguments. Gentzen found the Natural Deduction calculus unsuitable to prove his *Hauptsatz* and devised the Sequent Calculus to aide him. Gentzen [41] and Prawitz [71] gave translations from Natural Deduction proofs to Sequent Calculus proofs.

In contrast to the Natural Deduction Calculus, the Sequent Calculus provided a systematic and mechanical method for constructing a proof; this came at the cost of intuition and consequently the sequent calculus proofs are more difficult to comprehend as natural language arguments.

It is widely accepted that Natural Deduction is suited to the study of intuitionistic truths, while the Sequent Calculus is better suited to the study of classical truth. We recall that the two fundamental laws of truth for classical logics are:

- Law of the Excluded Middle: each formula is *true* or *false*.
- Law of Non-Contradiction: no formula is both *true* and *false*.

In this thesis, we study the Sequent Calculus for Classical Logic, and more specifically, we are interested in the extraction of computational content from proofs. We will therefore restrict forthcoming discussions to Classical Sequent Calculus.

Each sequent calculus we will consider in this thesis is a system that allows one to prove the validity of statements given in some formalised classical propositional

language. The statements of this language are propositional formulas as defined below. Subformulas are also defined as usual.

Definition 2.2.1 (Propositional Formula) *Propositional formulas are built up from (atomic) propositional variables that range over the countably infinite set* $\{\mathfrak{a}, \mathfrak{a}_1, \mathfrak{a}_2, \ldots\}$ *and represent basic propositions that may have only one of the truth-values from the set of all truth-values,* $\mathcal{T} = \{true, false\}$*. The set of propositional formulas* $\mathscr{F}$ *is ranged over by the infinite set* $\{A, A_1, A_2, \ldots\}$*. For* $n \geq 0$*, if* $A_1, \ldots, A_n$ *are propositional formulas, then the propositional connective* $\mathfrak{C}^n$ *with associated arity n may be used to construct the propositional formula* $\mathfrak{C}^n(A_1, \ldots, A_n)$*.*

Nothing else is a propositional formula.

For convenience, we will allow ourselves to use the standard notation for the connectives $\vee, \wedge, \rightarrow, \neg, \bot$ *and* $\top$*.*

Definition 2.2.2 (Subformula) *If A is a formula, then A is a subformula of A. For* $n \geq 0$*, if* $\mathfrak{C}^n(A_1, \ldots, A_n)$ *is a formula, then the subformulas of each of the formulas* $A_1, \ldots, A_n$ *are subformulas of* $\mathfrak{C}^n(A_1, \ldots, A_n)$*. Nothing else is a subformula.*

The Sequent Calculus (as opposed to the Natural Deduction Calculus) allows one to build arguments about *multiple cases* (conclusions) from a collection of assumptions. This is done by maintaining a set of open assumptions and a set of open conclusions at each step of the proof, encapsulating these details within a construct called a *sequent*. A sequent is of the shape '$\Gamma \vdash \Delta$' consisting of two parts or *contexts*: an *antecedent* Γ and a *succedent* Δ which represent (possibly empty) finite collections of propositional formulas. A suggested reading of the sequent,

$$\mathfrak{a}_1, \ldots, \mathfrak{a}_m \vdash \mathfrak{a}_{m+1}, \ldots, \mathfrak{a}_{m+n} \qquad (\textit{for } m, n \geq 0 \textit{ and } m+n > 0)$$

is "From the list of assumptions $\mathfrak{a}_1, \ldots, \mathfrak{a}_m$, it can be derived that not all of the cases $\mathfrak{a}_{m+1}, \ldots, \mathfrak{a}_n$ are impossible". A more common interpretation is the following $\mathfrak{a}_1 \wedge \ldots \wedge \mathfrak{a}_m$ *implies* $\mathfrak{a}_{m+1} \vee \ldots \vee \mathfrak{a}_{m+n}$.

The relation $\vdash$ defined between sets of formulas represents a logical consequence relation, defined as follows.

Definition 2.2.3 (Logical Consequence, $\vdash$) *The following conditions together specify Gentzen's basic notion of logical consequence (the last three conditions alone correspond to Tarski's basic definition of logical consequence).*

commutativity *: "The order of premises is irrelevant in any proof". For all formulas* A_1, A_2 *and all contexts* $\Gamma, \Gamma_1, \Gamma_2, \Delta, \Delta_1, \Delta_2$*:*

if $\Gamma \vdash \Delta_1, A_1, A_2, \Delta_2$ *(or* $\Gamma_1, A_1, A_2, \Gamma_2 \vdash \Delta$*) is provable,*
then $\Gamma \vdash \Delta_1, A_2, A_1, \Delta_2$ *(or* $\Gamma \vdash \Delta_1, A_2, A_1, \Delta_2$*) is provable.*

idempotency *: "The same consequences may be derived from many (or just one) premise or consequent". For all formulas* A *and all contexts* Γ, Δ*:*

if $\Gamma \vdash \Delta, A, A$ *(or* $A, A, \Gamma \vdash \Delta$*) is provable,*
then $\Gamma \vdash \Delta, A$ *(or* $A, \Gamma \vdash \Delta$*) is provable.*

monotonicity *: "No additional premise or consequent can affect the provability of a statement". For all formulas* A *and for all contexts* Γ, Δ*:*

if $\Gamma \vdash \Delta$ *is provable,*
then $A, \Gamma \vdash \Delta$ *and* $\Gamma \vdash \Delta, A$ *are provable.*

reflexivity *: "Every formula is deducible from itself". For all formulas* A*:* $A \vdash A$ *is provable.*

transitivity *: "Detours may be removed from any proof". For all formulas* A *and all contexts* Γ, Δ*:*

if $\Gamma \vdash \Delta, A$ *and* $A, \Gamma \vdash \Delta$ *are provable,*
then $\Gamma \vdash \Delta$ *is provable.*

In the following, we will write $\Gamma \vdash \Delta$ *when* Δ *can be deduced from* Γ *using the rules of* $\vdash$*.*

These properties are encoded as structural inference rules within Gentzen's Sequent Calculus, and are discussed in Section 2.2.1.

In Chapters 5 and 6 of this thesis, we will design and study a general system for building sequent calculi for propositional languages that employ arbitrary classical logical connectives as primitives. We will pay particular attention to the form of the inference rules that prescribe the use of the connectives in proofs. The generality of our approach requires us to be precise when formulating inference rules. We therefore introduce an additional level of abstraction over inference rules, which we call *rule schemes* or just 'schemes'. To avoid confusion in our presentations we will use different symbols and alphabets for each level of abstraction. We summarise our notation in the following definition.

Definition 2.2.4 (Proofs, Rules and Schemes) *A sequent represents some concrete*

statement made in some formal logic. The proof of a sequent details how one can infer the statement starting using only the basic axioms and inferences of the logic.

An inference rule describes, in general*, what can be inferred from other sequents according to the logic.*

A rule scheme describes valid shapes of inference rules.

To avoid confusion, we will adopt different notation for each level of abstraction; this is summarised below.

***Proofs** : Sequents mention only propositional formulas as defined in Definition 2.2.1 (e.g., $\mathfrak{a}$, $\complement^1(\mathfrak{a})$, $\complement^2(\mathfrak{a}_1, \complement^1(\mathfrak{a}_2))$).*

***Rules** : Rules mention variables over propositional formulas (denoted A) and variables for contexts (denoted Γ, Δ).*

***Schemes** : Schemes mention formula schemes (denoted $\mathscr{A}$) and context metavariables (denoted Ξ, Θ).*

Definition 2.2.5 (Inference Rules and Sequent-Schemes) *Every inference rule in the formulations of sequent calculus we will consider will be of the shape described by the following scheme.*

$$\frac{\bigsqcup_{k=1}^{m} \mathscr{A}_{1k} \;,\; \Xi_1 \vdash \Theta_1 \;,\; \bigsqcup_{k=(m+1)}^{n} \mathscr{A}_{1k} \qquad \dots \qquad \bigsqcup_{k=1}^{m} \mathscr{A}_{sk} \;,\; \Xi_s \vdash \Theta_s \;,\; \bigsqcup_{k=(m+1)}^{n} \mathscr{A}_{sk}}{\mathscr{A}_L \;,\; \bigsqcup_{j=1}^{s} \Xi_j \vdash \bigsqcup_{j=1}^{s} \Theta_j \;,\; \mathscr{A}_R} \; (R)$$

where:

- $s \geq 0$, $m, n \geq 0$ *and* $(m+n) > 0$.
- *The symbols* $\Xi, \Xi_1, \Xi_2, \dots$ *and the symbols* $\Theta, \Theta_1, \Theta_2, \dots$ *are (variables for) contexts and* $\mathscr{A}_L, \mathscr{A}_R, \mathscr{A}, \mathscr{A}_1, \mathscr{A}_2, \dots$ *are (variables for) propositional formulas.*
- *Regarding the variables* $\mathscr{A}_L$ *and* $\mathscr{A}_R$*,* one *of the following cases holds for each inference rule:*
  - *Either* $\mathscr{A}_L$ *or* $\mathscr{A}_R$ *is exclusively present.*
  - *If* $\mathscr{A}_L$ *and* $\mathscr{A}_R$ *are both present, then* $\mathscr{A}_L = \mathscr{A}_R$ *and* $s = 0$.
  - *Neither* $\mathscr{A}_L$ *nor* $\mathscr{A}_R$ *is present, in which case* $s > 0$.
- *The notation* $\bigsqcup_{j=1}^{k} X_j$ *is shorthand for* $X_1, X_2, \dots, X_k$.
- *The comma-symbol ',' is an overloaded abstract operation that specifies how to combine instances of: (i) context metavariables (ii) formula schemes and (iii) context metavariables with formula schemes.*

A sequent-scheme *is a pair whose components are lists of context metavariables and formula schemes, which will be written as* $\mathscr{A}_1, \ldots, \mathscr{A}_m, \Xi_1, \ldots, \Xi_t \vdash \Theta_1, \ldots, \Theta_t, \mathscr{A}_{m+1}, \ldots, \mathscr{A}_n$, *for* $t \geq 0$, $m, n \geq 0$ *and* $m > n$.

The sequence of sequent schemes above the horizontal line will collectively be referred to as the premises. *The sequent scheme below the line is the* conclusion. *The bracketed string of symbols to the right of the horizontal line is the rule name—'R' in the scheme above.*

The formulas which occur in the rule premises and not in the conclusion are the component-formulas *that are* bound *by the rule.*

An application of an inference rule with s premises to a collection of s sequents, proceeds by building a mapping by matching each variable in each rule premise to the appropriate parts of each sequent. If there is no match, the rule is not applicable to those sequents. If a mapping does exist, a horizontal line is drawn beneath the sequents (arranged in sequence as shown), under which the image under the found mapping of each variable in the rule conclusion is written. A *proof* of a statement is then a derivation tree rooted at that statement (called the *endsequent*) with instances of the axiom rule at its leaves; all other nodes in the tree are built using instances of inference rules.

Inference rules have traditionally been grouped into *structural rules*, *cut*, *axiom* and *logical rules*. These are discussed in the following sections.

2.2.1 Structural Rules

The structural rules are independent of any object language and dictate the "shape" of valid arguments; they are responsible for describing how one may collect premises and conclusions (rather than how statements are constructed).

Gentzen formalised the properties of his logical consequence relation (i.e., commutativity, idempotency, monotonicity, reflexivity and transitivity) with five structural rules (respectively) called *exchange*, *contraction*, *weakening*, *axiom* and *cut*. The exclusion of any of these rules would fundamentally change the logic in which one was reasoning (such logics are commonly referred to as substructural

logics). The first three of these rules are given below.

$$\frac{\Gamma_1, A_1, A_2, \Gamma_2 \vdash \Delta}{\Gamma_1, A_2, A_1, \Gamma_2 \vdash \Delta}\,(Exchange_L) \qquad \frac{\Gamma \vdash \Delta_1, A_1, A_2, \Delta_2}{\Gamma \vdash \Delta_1, A_2, A_1, \Delta_2}\,(Exchange_R)$$

$$\frac{A, A, \Gamma \vdash \Delta}{A, \Gamma \vdash \Delta}\,(Contraction_L) \qquad \frac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A}\,(Contraction_R)$$

$$\frac{\Gamma \vdash \Delta}{A, \Gamma \vdash \Delta}\,(Weakening_L) \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A}\,(Weakening_R)$$

Viewed as a syntactic operation on a proof, an exchange swaps two adjacent formulas within a sequence, a contraction deletes an adjacent duplicate formula from the extremity of a sequence, and a weakening appends an arbitrary formula to the end of the sequence. Gentzen needed explicit formulations of these rules in his sequent calculi because of his choice of representation of contexts as ordered lists (or *sequences*) of formulas. Since the intended rôle of this formalism is to show provability, an arguably unrequired side-effect of this representation is the ability to construct a number of proofs for a statement that differ only in structure. Kleene, [58], devised several variants of Gentzen's Sequent Calculus which greatly simplified the presentation of proofs; his sequent calculus named G3 treated cedents as *sets* of formulas, making the explicit exchange and contraction rules obsolete (we note that these rules are only made implicit and not eliminated). He was also able to absorb the explicit weakening rules in his G3 system by allowing arbitrary formulas in axioms, i.e. with the following formulation:

$$\frac{}{A, \Gamma \vdash \Delta, A}\,(Ax)$$

Kleene's modifications, which have been widely adopted, optimize proof search by minimizing the choices of premises for a given conclusion. We remark however that they do not eliminate all structural proof permutations, for such proof permutations are a natural feature of the Sequent Calculus and cannot be removed entirely (or at least not without working very hard). These permutations arise from the freedom to manipulate multiple assumptions and conclusions within a proof and from the symmetry of the left and right inference rules.

A special rule of inference called *Cut* is often employed in sequent calculi. In this thesis, we will design our sequent calculi in the style of Gentzen so that this rule is *admissible*—in the sense that every proof may be transformed into a cut-free

proof of the same endsequent.

$$\frac{\Gamma_1 \vdash \Delta_1, A \qquad A, \Gamma_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\,(Cut)$$

We identify the instance of the formula-variable bound by the rule (in this case the instance of A) as the *cut formula*.

The cut rule captures the notion of a lemma in proof theory: that a proof of some statement can be realised via a detour through some intermediate result. This is especially useful, since deducing theorems from first principles is time-consuming and verbose. To give an idea of the conciseness the cut allows one to achieve, in the worst case, a proof which utilises the cut rule can grow hyper-exponentially in size when expressed as a cut-free proof. A further use of the cut is in providing a straightforward correspondence with Natural Deduction style proofs.

Some attention should be paid to how contexts are handled in branching inference rules (those rules which have more than one premise). An *additive* formulation (see $\vee_L$ in Section 2.2.2 for an example) shares contexts across the rule premises, while a *multiplicative* rule (such as the cut rule shown above) would join the contexts. The choice does not affect provability and preference is dependent upon one's aims. For example, an automated proof search tool might favour an additive formulation, while one seeking compact proofs will likely prefer multiplicative rules.

Kleene was able to capture Gentzen's notion of logical consequence with a formulation of sequent calculus featuring implicit contraction, exchange and weakening rules; he called this calculus G3a. We give the most basic fragment, featuring no logical connectives, below.

Definition 2.2.6 (Basic fragment of G3a, G3A-BASIC) *The fragment of Kleene's G3a sequent calculus without any logical connectives (which we will refer to as* G3A-BASIC*) is given by the following rules.*

$$\frac{}{A, \Gamma \vdash \Delta, A}\,(Ax) \qquad \frac{\Gamma_1 \vdash \Delta_1, A \qquad A, \Gamma_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\,(Cut)$$

where,

- $A, A_1, A_2, \ldots$ *are variables for formulas.*
- $\Gamma, \Gamma_1, \Gamma_2, \ldots$ *and* $\Delta, \Delta_1, \Delta_2, \ldots$ *are context variables.*

- *The comma is an abstraction operation which maps to the set union of propositional formulas in proofs.*
- *The comma is overloaded and also used as a shorthand:* $\Gamma, A = \Gamma, \{A\}$.

2.2.2 Logical Rules

In the Sequent Calculus, the connectives of a formal propositional language are *defined*[1] by a collection of logical inference rules describing the cases when a connective may be introduced to the antecedent and succedent parts of a sequent in the construction of a proof. For example, the following pair of inference rules define the cases when the connective $\vee$ may be introduced.

$$\frac{A_1, \Gamma \vdash \Delta \quad A_2, \Gamma \vdash \Delta}{A_1 \vee A_2, \Gamma \vdash \Delta}\,(\vee L) \qquad \frac{\Gamma \vdash \Delta, A_1}{\Gamma \vdash \Delta, A_1 \vee A_2}\,(\vee R_1) \qquad \frac{\Gamma \vdash \Delta, A_2}{\Gamma \vdash \Delta, A_1 \vee A_2}\,(\vee R_2)$$

The new formula introduced beneath the horizontal line is the *principal formula*.

Ketonen, in [57], studied alternate formulations of inference rules focusing on proof search. Gentzen originally used two left conjunction and two right disjunction rules (shown above) in the proof of his *Hauptsatz* in order to highlight the similarities between intuitionistic and classical logics when embedded within the Sequent Calculus (see [50] for details). Ketonen, making use of the structural properties of the calculus, proposed *invertible* reformulations of the rules for logical connectives noting the advantage gained in proof search: that in searching for a derivation of some statement, a single unique rule is applicable at each stage of the construction.

Definition 2.2.7 (Invertibility) *A rule of inference for a logical connective is* invertible *if derivability of the lower sequent implies derivability of the upper sequent.*

The two right-introduction rules for disjunction shown above, for example, can be reformulated as an invertible rule as follows.

$$\frac{\Gamma \vdash \Delta, A_1, A_2}{\Gamma \vdash \Delta, A_1 \vee A_2}\,(\vee R)$$

In this thesis, we will study connectives that are defined by invertible logical rules.

[1] See Section 2.2.4 for an explanation of the sense in which this word is used.

Since we will be investigating arbitrary logical connectives in the sequent calculus, we will take some time to formalise a general scheme for building logical inference rules (which we recall are themselves schemes for building proofs).

Each of the inference rules for the propositional connectives in Gentzen's sequent calculi builds a complex formula out of less complex formulas.

Definition 2.2.8 (Subformula Property) *Each formula occurring in any sequent of a cut-free proof is a subformula of some formula occurring in the endsequent.*

For the sequent calculi we will consider, we will also require that our cut-free proofs have this subformula property (see Section 2.2.4 for further justification).

2.2.3 Cut-Elimination

According to Gentzen's main result, his *Hauptsatz*, every application of the cut rule in a derivation (constructed using his formulations of sequent calculus) can either be replaced by simpler[2] instances or be removed from the proof. The rules which prescribe this transformation are known as cut-elimination rules. The main idea of Gentzen's procedure was to apply *local* proof transformation rules to a derivation that had the effect of shifting the cut upwards towards the leaves. Once at a leaf, a cut is easily removed by considering a finite number of base cases. In this Section, we will look at the ideas behind this procedure in more detail.

When a cut is not at the leaves of a derivation, it can take on two forms depending on its position in the proof. Consider the instance of the cut rule with cut formula $A_1 \rightarrow A_2$ corresponding to logical implication[3] shown below.

$$\dfrac{\dfrac{\dfrac{\boxed{M}}{A_3, A_1 \vdash A_2, A_4}}{A_3 \vdash (A_1 \rightarrow A_2), A_4}(C_R)}{\vdash (A_1 \rightarrow A_2), (A_3 \rightarrow A_4)}(C_R) \qquad \dfrac{\boxed{P} \quad \boxed{Q}}{\vdash A_1 \quad A_2 \vdash}(C_L)}{\vdash (A_3 \rightarrow A_4)}(Cut)$$

[2] His measure calculated a *rank* for each cut, based on the number of arguments in the cut formula and the height of the cut.

[3] Incidentally, this would be represented as $\complement^2_{1101_2}(A_1, A_2)$ in our syntax, but we refrain from using this notation until after its definition in Definition 2.2.11.

The elimination of a cut (in the absence of weakening) involves removing *all* information about a cut formula $(A_1 \rightarrow A_2)$ from the proof. Recall that an application of the cut rule in a proof represents a detour step of a proof; eliminating the cut corresponds to building a more direct argument. This direct argument will not mention any information used in the detour, and so the arguments of the cut formula (in the example, A_1 and A_2) will also need to be eliminated. In the proof above, the cut (at its current location) cannot locally access the rules in which these arguments are discharged, since they are further up in the derivation. The cut is therefore 'pushed' upwards through the structure of the derivation towards the relevant positions. Such an instance of a cut is referred to as a *commuting cut*. A typical proof transformation rule applicable in this situation would result in:

$$\dfrac{\dfrac{\dfrac{\boxed{M}}{A_3, A_1 \vdash A_2, A_4}\,(C_R)}{A_3 \vdash (A_1 \rightarrow A_2), A_4} \qquad \dfrac{\boxed{P}\quad\boxed{Q}}{\vdash A_1 \quad A_2 \vdash}\,(C_L)}{(A_1 \rightarrow A_2) \vdash}\,(Cut)}{\dfrac{A_3 \vdash A_4}{\vdash (A_3 \rightarrow A_4)}\,(C_R)}$$

This instance of the cut is called a *logical cut*, and the subformulas of the cut formula are discharged in the preceding proof steps by the appropriate inference rules for the logical connective. We will say the cut formulas are *introduced* when the child sequents of the cut rule are either axioms or logical inference rules whose principal formulas are those cut formulas. In this case, an appropriate proof transformation rule would eliminate the cut (and cut formulas) from the proof and possibly form several new cuts between subformulas of the cut formula. One possible transformation is:

$$\dfrac{\dfrac{\dfrac{\boxed{P}}{\vdash A_1} \quad \dfrac{\boxed{M}}{A_3, A_1 \vdash A_2, A_4}\,(Cut)}{A_3 \vdash A_2, A_4} \qquad \dfrac{\boxed{Q}}{A_2 \vdash}\,(Cut)}{\dfrac{A_3 \vdash A_4}{\vdash (A_3 \rightarrow A_4)}\,(C_R)}$$

We will call the rule that describes this transformation step a *logical-cut reduction rule*, and providing a method of deriving all 'good' reduction rules for classical logical connectives in general is the main contribution of this paper. Observe that the cuts, although greater in number, have simpler cut formulas.

There are two other cases to consider: when the cut formula is weakened and when it is the result of an axiomatic formula. The cut-elimination steps for these

cases are:

$$\dfrac{\dfrac{}{A_2 \vdash A_2, A_1}\,(Ax) \qquad \boxed{N}\ A_1, \Gamma \vdash \Delta}{\Gamma, A_2 \vdash A_2, \Delta}\,(Cut) \quad \Rightarrow \quad \dfrac{}{\Gamma, A_2 \vdash A_2, \Delta}\,(Ax)$$

$$\dfrac{\dfrac{}{\Gamma, A_1 \vdash A_1, \Delta}\,(Ax) \qquad \boxed{N}\ A_1, \Gamma \vdash \Delta}{A_1, \Gamma \vdash \Delta}\,(Cut) \quad \Rightarrow \quad \boxed{N}\ A_1, \Gamma \vdash \Delta$$

2.2.4 On the Importance of Cut-Elimination

Aside from obtaining the consistency of a logic as a corollary of cut-elimination, the cut-elimination theorem plays an important rôle in giving proof-theoretical semantics to sequent calculus logics. Gentzen's remark on the autonomy of introduction rules[4], that they give the full 'definition' of a logical connective in a proof-theoretic sense, has been adopted and developed upon by many authors [77, 39]. We will give a brief summary of these works in the following.

Advocates of the analytically valid view of logical connectives, so called anti-realists, maintain that the structure of the inference rules are entirely responsible for giving a connective its meaning. Those opposing this view argue that more is needed than mere structural rules to give meaning; Prior is famous for his controversial connective ♣ (pronounced 'tonk') [72] whose sequent rules are given below[5].

$$\dfrac{\Gamma \vdash \Delta, A_1}{A_1 \clubsuit A_2, \Gamma \vdash \Delta}\,(\clubsuit L) \qquad \dfrac{A_2, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, A_1 \clubsuit A_2}\,(\clubsuit R)$$

Prior argues that his 'definition' of ♣ is perfectly acceptable from a proof-theoretic perspective since there should be no extra requirement to test whether the introduction rules are valid. However, ♣ can reduce a logic to nonsense (allowing one to prove *any* statement from any two unrelated proofs) and therefore, as Prior argued, something more than the structure of introduction rules must be required to give a connective meaning.

A number of responses to Prior's attack stem from Belnap's initial response, [17], whose key observation was that in defining a logical connective, one is not working from first principles. The turnstile, '$\vdash$', denotes a logical consequence relation, and so any extension of the logic should preserve this relation—if one wishes to

[4] Originally, in the context of Natural Deduction, but which carries over to the Sequent Calculus in a straightforward way.

[5] Prior works in a Natural Deduction system. The analogous sequent calculus rules shown are from [20].

continue to build deductive arguments in the spirit of the original logic. Taking this view, the rules for ♣ are perfectly valid, except that in a system with such rules one can no longer claim to be reasoning in the original logic. This prompts the question of what kinds of inference rules can define 'good' logical connectives that preserve the properties of the original logic. Hacking and Dummett, as we will see, are two influential researchers whose works have provided some answers to this question.

Hacking who is interested in Classical Sequent Calculus, argues that one can preserve the logical consequence relation with the requirement that any extension is *conservative* and that the corresponding inference rules for the connective have the *subformula* property [45].

Definition 2.2.9 (Conservative Extension, [77]) *An extension of a logic S on a language L by the addition of a constant $\complement^n$, yielding a language L′ extending L and a system S′ extending S containing rules for the use of $\complement^n$, is conservative if any inference in L provable in S′ (i.e., provable in the extension but not containing the new vocabulary $\complement^n$) is provable in S.*

Remark 2.2.10 *The addition of ♣ to* G3A-BASIC *does not yield a conservative extension (of* G3A-BASIC*), since we now can build a proof of $\Gamma \vdash \Delta$ (a statement expressed purely in the syntax of* G3A-BASIC*) via a detour through ♣ given proofs of $\Gamma \vdash \Delta, A_1$ and $A_2, \Gamma \vdash \Delta$ (both of which are also expressed in the syntax of* G3A-BASIC*).*

The requirement of conservativeness guarantees that the set of provable statements in the original system are not altered by the addition of any new connective. The subformula property then ensures that the introduction rules for the connective are only a **recursive** extension of the original notion of logical consequence (i.e. that any connective can be expressed purely in terms of the original unaltered relation).

Hacking observes that one way of guaranteeing a conservative extension is to require a cut-elimination theorem. This follows intuitively, since any proof of a statement in the original syntax that makes a detour through the new connective can be replaced with a direct proof without that detour (thus the use of the new connective is shown to be redundant in proofs of statements made in the original syntax).

Dummett who has different motivations for his investigations, also takes a proof-theoretic view of logic, though he works on natural deduction calculi and does

not require such a strong condition as conservativeness [36]. Instead, he allows only logical connectives defined by rules that are *harmonious* and have the subformula property. Dummett, in fact, gives two notions of harmony: total harmony and intrinsic harmony. The former relates to conservativeness, while the latter relates to normalisation (which corresponds to cut-elimination in sequent calculi). A pair of inference rules are *intrinsically harmonious* if the introduction and elimination rules are related so that one can draw from an assertion of a proposition whose form displays the connective *only* those grounds which were needed to establish that assertion.

We note that, although the ♣-connective passes the test for the subformula property, it could not be classed as a logical connective according to Dummett, since a proof detour through ♣ cannot be eliminated when it is introduced as the cut formula, i.e. it is not defined by harmonious rules.

2.2.5 Axiomatisation of Logical Connectives

Model-theoretic semantics for the propositional connectives of classical logic can be obtained from a truth-function. In this sense, a truth-function *defines* a classical logical connective. Such truth-functions are commonly expressed as *truth tables*.

Definition 2.2.11 (Truth Table $\mathbb{C}_i^n$) *A* truth table*, written $\mathbb{C}_i^n :: [\mathcal{T}] \to \mathcal{T}$ for a logical connective of arity n is a function which maps a* list *of truth values to a truth value.*

The function $\mathbb{C}_i^n$ can be visualised as a table with 2^n rows (indexed $0 \ldots 2^n{-}1$) and $n{+}1$ columns. The first k columns are labelled $A_1, \ldots, A_n$ and the $(k{+}1)^{th}$ column (or 'defining' column) is labelled $\complement_i^n(A_1, \ldots, A_n)$. The truth value in row r of the defining column is $\ddot{n}_i[r]$ (where $0 < r \leq 2^n{-}1$).

Pictorially, we have:

	A_1	A_2	A_3	$\ldots$	A_n	$\complement_i^n(A_1, \ldots, A_n)$
0	0	0	0	$\ldots$	0	$\ddot{n}_i[0]$
1	0	0	0	$\ldots$	1	$\ddot{n}_i[1]$
				$\vdots$		
$2^n{-}1$	1	1	1	$\ldots$	1	$\ddot{n}_i[2^n{-}1]$

Definition 2.2.12 (Truth Function) *A* valuation *is an assignment of truth values to propositional variables. Given a valuation* ϕ*, a* truth function $\langle\!|\cdot|\!\rangle_\phi :: \mathscr{F} \to \mathcal{T}$ *under that valuation maps a propositional formula to a single truth value. The interpretation is defined inductively on the structure of the propositional formula A with respect to a valuation function,* ϕ*:*

1. *if* $A \equiv \mathfrak{a}$*, then* $\langle\!|A|\!\rangle_\phi = \phi(\mathfrak{a})$.
2. *if* $A \equiv \complement_i^n(A_1, \ldots, A_n)$*, then* $\langle\!|A|\!\rangle_\phi = \mathbb{C}_i^n[\ \langle\!|A_1|\!\rangle_\phi, \ldots, \langle\!|A_n|\!\rangle_\phi\ ]$.

Definition 2.2.13 (True Arity) *A logical connective* $\complement^n$ *of arity n has propositional arguments* $\mathfrak{a}_1, \ldots, \mathfrak{a}_n$*. An argument* $\mathfrak{a}_j$ *of a connective (with* $0 < j \leq n$*), is said to be* trivial *if-and-only-if, for* all *possible valuations* ϕ*:*

$$\langle\!|\complement_i^n(\mathfrak{a}_1, \ldots, \mathfrak{a}_{j-1}, \complement_1^0, \mathfrak{a}_{j+1}, \ldots, \mathfrak{a}_n)|\!\rangle_\phi \equiv \langle\!|\complement_i^n(\mathfrak{a}_1, \ldots, \mathfrak{a}_{j-1}, \complement_0^0, \mathfrak{a}_{j+1}, \ldots, \mathfrak{a}_n)|\!\rangle_\phi$$

The true arity *of a connective is then a count of its non-trivial arguments.*

Intuitively, a connective whose true arity is not the same as its arity is one whose truth-function always ignores one or more of its arguments in the computation of its truth value. For example, consider an arity 2 negation function which only negates the truth value of its first argument always ignoring the second; it's 'true arity' is 1.

Example 2.2.14 (Truth table for the connective $\complement^2_{1101_2}$**)** *The truth function* $\mathbb{C}^2_{1101_2}$ *for a binary connective* $\complement^2_{1101_2}$ *(commonly written* $\rightarrow$*) denoting logical implication is defined as follows.*

	A_1	A_2	$A_1 \rightarrow A_2$
0	0	0	1
1	0	1	1
2	1	0	0
3	1	1	1

The truth function enumerates all possible assignments of truth values to the connective's arguments A_1 *and* A_2*.*

With these semantics, one may test the validity of an arbitrary statement $\Gamma \vdash \Delta$ via a truth table construction, reading the turnstile as an implication, the commas in the antecedent as conjunctions, and the commas in the succedent as disjunctions.

Example 2.2.15 (Testing Validity using Truth tables) *Consider the question: is A_1 derivable from $A_1 \rightarrow A_2$ and A_2 (i.e., does $A_1 \rightarrow A_2, A_2 \vdash A_1$ hold). This is equivalent to testing the validity of the formula $((A_1 \rightarrow A_2) \wedge A_2) \rightarrow A_1$. We proceed by considering all possible assignments of truth-values to the propositional formulas A_1 and A_2, and incrementally build up the truth-values of all subformulas of $((A_1 \rightarrow A_2) \wedge A_2) \rightarrow A_1$. The statement is valid if the truth-value assigned to the formula $((A_1 \rightarrow A_2) \wedge A_2) \rightarrow A_1$ is always 1.*

Incrementally building up the formula in question from its subformulas, and accumulating this information together in a table, we get:

A_1	A_2	$A_1 \rightarrow A_2$	$(A_1 \rightarrow A_2) \wedge A_2$	$((A_1 \rightarrow A_2) \wedge A_2) \rightarrow A_1$
0	0	1	0	0
0	1	1	1	1
1	0	0	0	1
1	1	1	1	1

Thus, the formula is not valid since it has a truth-value of 0 when A_1 and A_2 are assigned the truth-value 0.

Various researchers have shown that one can extract sequent calculus style inference rules directly from truth tables in a mechanical fashion; we will briefly discuss those most relevant to our work.

Call, in [24], describes such a mechanical procedure for classical propositional connectives. In his paper, he (informally) outlines an interesting two-phase algorithm for building sequent calculus rules from truth tables. The first phase builds a pair of inference rules for a connective by mapping each row of the truth table to a rule premise. The second phase simplifies each inference rule by a pairwise merging of premises. He extends a basic sequent calculus with the generated inference rules for propositional formulas, then adopts a procedure by Kleene [59] (which he calls the 'Kleene Search Procedure') to show that the resulting sequent calculus yields the valid formulas of the propositional logic. Our study of Call's algorithm has led us to several deep insights into the relation between classical truth tables and the inference rules for propositional connectives in the sequent calculus. These insights have helped us in the development of our own contributions, so we will spend some time formalising Call's algorithm in quite some detail. This work is presented in Section 6.1.2.

Using a different approach, Baaz *et al.* [6, 7] describe their tool called MULTLOG (implemented in Prolog) which also mechanically builds simplified sequent cal-

culus inference rules from truth tables. However, their methods are more general than Call's, and can deal with a wider class of logics, namely first-order and many-valued logics. Their method of extraction of inference rules from truth tables also differs. The authors use an adapted Quine-McCluskey procedure[6] to express the raw formulas extracted from each line of the truth tables as a conjunction of disjunctions. They explain that in this form, "the expressions are minimal in the number of conjuncts and the number of disjuncts per conjunct", so the number of premises per sequent inference rule are also minimal. A Natural Deduction presentation of the resulting propositional logic is also given.

Most interestingly, the authors give details on how one can obtain a *local* cut-elimination procedure that successfully reduces a cut when the cut formula is built from the introduction rules for the generated connectives. As they explain, the key component in defining this procedure (and indeed in obtaining a cut-elimination theorem) is obtaining a function (which they call *Red*) that is able to eliminate the outermost logical symbol from the cut formula, and thus reduce the degree of the cut. The technique they use to build this function *Red* is based on many-valued resolution techniques [5]. We will briefly summarize this technique, but adapt it to the more familiar setting of classical logic.

Baaz *et al.* start from a truth table definition of a connective. The left- (right-) introduction rule is built from the rows of the truth table where the principal formula is assigned a truth-value of 0 (1). The function *Red* takes as its input the pair of inference rules. The output is generated as follows. *Clauses* (sets of literals[7]) are extracted from the rule premises. Each clause extracted from a particular rule corresponds to a case when the connective is assigned a particular truth-value, e.g. all clauses extracted from the premises of the left introduction rule correspond to the cases when the connective is assigned the value 0 in the truth table—for this is how the premises were built. If one builds a clause $\mathcal{C}$ by combining the premises of all introduction rules, that clause is unsatisfiable, since a connective cannot have more than one truth-value (according to the law of non-contradiction for classical logic). Many-valued resolution [5] is refutation complete (the empty clause is derived from any inconsistent set of clauses), and so there is a *resolution deduction*[8] of the empty clause from the set $\mathcal{C}$. Once a resolution refutation is found, it is mapped to a deduction schema, which corresponds to the output of the function *Red*.

[6] also known as 'the method of prime implicants', used for minimization of Boolean functions, equivalent to Karnaugh mapping—but more practical for higher arity connectives.

[7] A literal is an atomic formula with a truth-value; e.g. the literal A^0 means A is false, and the literal A^1 means A is true.

[8] A resolution deduction is a deduction built from applications of the resolution rule.

Example 2.2.16 (Extracting the function *Red*) *Consider the sequent calculus inference rules for the disjunction connective.*

$$\frac{A_1, \Gamma_1^L \vdash \Delta_1^L \qquad A_2, \Gamma_2^L \vdash \Delta_2^L}{A_1 \vee A_2, \Gamma_1^L, \Gamma_2^L \vdash \Delta_1^L, \Delta_2^L}\,(\vee L) \qquad \frac{\Gamma_1^R \vdash \Delta_1^R, A_1, A_2}{\Gamma_1^L, \Gamma_2^L \vdash \Delta_1^L, \Delta_2^L, A_1 \vee A_2}\,(\vee R)$$

The clause translation of all the premises is:

$$\mathcal{C} = \{\{A_1^0\}, \{A_2^0\}, \{A_1^1, A_2^1\}\}$$

A possible refutation is:

$$\dfrac{\dfrac{\{A_1^1, A_2^1\} \quad \{A_1^0\}}{\{A_2^1\}} \qquad \{A_2^0\}}{\emptyset}$$

From this, one can obtain a deduction schema, by replacing the resolution rule with the cut rule and translating the sets of clauses to sequent-schemes. For example, the refutation above would yield the following derivation scheme:

$$\dfrac{\dfrac{\Gamma_1^R \vdash \Delta_1^R, A_1, A_2 \quad A_1, \Gamma_1^L \vdash \Delta_1^L}{\Gamma_1^R, \Gamma_1^L \vdash \Delta_1^R, \Delta_1^L, A_2}\,(Cut) \qquad A_2, \Gamma_2^L \vdash \Delta_2^L}{\Gamma_1^L, \Gamma_2^L, \Gamma_1^R \vdash \Delta_1^L, \Delta_2^L, \Delta_1^R}\,(Cut)$$

The above schema would be the output of the function Red *for the input pair of rules* $(\vee L)$ *and* $(\vee R)$.

Ciabattoni and Leitsch, in [30], also study the automation of building cut-elimination procedures, but for single-conclusion sequent calculus systems (actually for 'knotted commutative calculi'). Their procedure is also based on resolution techniques.

We would like to make two key observations about these two systems which search for the key cut-elimination rule for the logical connective. First, they employ an (inefficient) brute force technique to find the rule. This does not scale well when studying connectives of higher arity. Second, they are designed to find only one of the possibly many permutations of the cut-elimination rule. This is acceptable in the proof-theoretic setting since the permutation of a proof is semantically unimportant—the important feature is that a proof *exists*. As we will see in the next section, there are settings where *each* permutations can become important.

2.3 Computability Theory

Computability Theory is primarily concerned with the study of *computable functions*; informally, these are functions whose values can be found mechanically by following a sequence of atomic instructions and given unbounded resources of time and storage space. In 1936, Turing, Church and Kleene showed that not all functions can be solved in this way (this even includes the use of significantly more powerful computers such as quantum computers). Despite this negative result, the class of computable functions is very rich.

Turing is well-known for his design of a powerful abstract machine, known as "The Universal Turing Machine", conjectured to capture the human notion of what is computable. Each particular 'Turing Machine' describes a computable function or 'algorithm' at a very low-level of granularity. The machines themselves are inherently imperative and also very easily realised in hardware; it was these features that shaped the field of computing in the years that followed.

Around the same time, Church, with the help of Kleene and Rosser, had formed his own notions to capture the class of computable functions. He presented his pure (and untyped) λ-calculus with its reduction-theory in [28] focusing on computability. We will review the λ-calculus in the following section.

2.3.1 Review of λ-calculus

The λ-calculus was the result of Church's attempt to build a formal logical foundation for mathematics based on the notion of functions. When his work was shown to be inconsistent (by admitting a variant of Richard's Paradox) in 1933, he extracted out the consistent part essentially by removing any axioms related to logical notions. What remained was a very succinct language for describing functions, i.e., via abstraction, application and the process of substitution. We give the formal description of his language below.

Definition 2.3.1 (Pure Untyped λ-Calculus) *The language for the untyped λ-calculus is defined by the following syntax, where $x, y, z, \ldots$ range over the infinite set of variables and $M, N, \ldots$ range over λ-terms.*

$$\begin{array}{lccccc} M, N ::= & x & | & (\lambda x.M) & | & (MN) \\ & \textit{variable} & & \textit{abstraction} & & \textit{application} \end{array}$$

Applications associate to the left as usual, and we will allow ourselves to omit bracketing when there is no possibility of confusion.

The abstraction term gives the basic structure of a function. The "λ" symbol distinguishes the variable, x, as the formal parameter. Multi-argument functions can be modelled by composing abstractions. The "." separates the formal parameter from the function body. There is no extra symbol to denote application—it is simply the juxtaposition of two λ-terms.

Formal parameters are placeholders for expressions. To define the method of computation over λ-terms, it will become necessary to distinguish between the local variables of a function (i.e., those variables which refer to, or are *bound* to, a formal parameter) and the variables which do not refer to a formal parameter, called *free* variables. The following definition on λ-terms serves to make this distinction.

Definition 2.3.2 (Free and Bound Variables in λ-terms) *The sets of* free *and* bound *variables and of a λ-term M, denoted $fv(M)$ and $bv(M)$, respectively, are defined recursively over the structure of λ-terms.*

$$\begin{array}{lll} fv(x) & = \{x\} & bv(x) & = \emptyset \\ fv(\lambda x.M) & = fv(M) \setminus \{x\} & bv(\lambda x.M) & = \{x\} \cup bv(M) \\ fv(MN) & = fv(M) \cup fv(N) & bv(MN) & = bv(M) \cup bv(N) \end{array}$$

Terms which have no free variables are called closed *terms.*

In observing that the purpose of free and bound variables is to encode the relationship between a formal parameter and its use in the function body, it is clear that the name of the parameter itself is irrelevant. An equivalence between terms that differ solely on the names of formal parameters is defined as follows.

Definition 2.3.3 (α-equivalence of λ-terms) *Two λ-terms M and N are said to be* α-equivalent, *written $M \equiv_\alpha N$, if one is obtainable from the other by renaming bound variables.*

The computational rule of the calculus, the β-reduction rule, describes how one may compute the value of a function for a given input. An application of an abstraction to some other term, written $(\lambda x.M)N$, is called a reducible expression or *redex*. The process of evaluating such a redex involves substituting a copy of

the argument for each occurrence of the free variable in the function body that refers to the function's formal parameter.

Definition 2.3.4 (β-reduction) *The key computational rule of the λ-calculus is,*

$$(\beta): \quad (\lambda x.M)N \rightarrow M\{N/x\}$$

The term $M\{N/x\}$ is *the term M where the term N has been substituted for each occurrence of the free variable x. We emphasise that the notation $\{N/x\}$ is a meta-operation, not part of the language of λ-terms; this operation could be defined by:*

$$\begin{array}{lll} x\{N/x\} & = N & \\ y\{N/x\} & = y & \text{if } y \neq x \\ (M_1 M_2)\{N/x\} & = M_1\{N/x\}M_2\{N/x\} & \\ (\lambda y.M)\{M/x\} & = (\lambda y.M) & \text{if } y = x \\ (\lambda y.M)\{M/x\} & = (\lambda y.M\{N/x\}) & \text{if } y \neq x \end{array}$$

In Section 2.4, we will look at various implementation techniques which have been proposed to compute this substitution.

The β-reduction relation was shown to be *confluent* in [29]. Although there may be many redexes in any particular λ-term, the confluence property guarantees that the same result can be computed regardless of the order in which the redexes are chosen for evaluation. A λ-term that has no redexes is said to be in *normal form*.

In the untyped λ-calculus one is allowed to apply any two terms. In particular, a value can be applied to a function (e.g. 5 *cos*), suggesting some form of restriction should be placed on the structure of terms. Since mathematical functions are being modelled, a seemingly good restriction would be to insist on the constraints that would normally apply to mathematical functions. An informal description of these is given below.

1. All variables are members of some well defined set.
2. Instances of abstraction terms $(\lambda x.M)$ are (anonymous) functions whose domain and range are well defined sets. The function maps an input value from its domain to an output value in its range.
3. All instances of applications (MN) are function applications. That is, the left term M should always be treated as a function, and the right term N as an argument whose value is in the domain of the function. The application should produce a result in the range of the function.

These constraints are formalised by a *type system*. A suitable and fairly simple language of types for the λ-calculus is given below.

Definition 2.3.5 (Types) *Types, denoting non-empty sets of values, are ranged over by* $A, A_1, A_2, \ldots$ *and are defined over a set of type variables* $\phi, \phi_1, \phi_2, \phi_3, \ldots$. *The set of types is constructed by the following grammar.*

$$A ::= \phi \mid A \to A$$

Two different ways of extending the λ-calculus with a type-theory were proposed by Church [27] and Curry [33]. Church's approach was to make the type annotations part of the syntax, yielding a typed language. His simply-typed λ-calculus is formulated by the grammar given below.

$$M, N ::= x^A \mid (\lambda x^{A_1}.M^{A_2})^{A_1 \to A_2} \mid (M^{A_1 \to A_2} N^{A_1})^{A_2}$$

We point out that, in the typed-language, expressions such as $(\lambda x.x)y$ are no longer valid. The appropriate form would be $((\lambda x^A.x^A)^{A \to A} y^A)^A$. Notice that the term $(\lambda x.xx)(\lambda y.yy)$ cannot be annotated with types and is therefore omitted from the language.

Curry's approach, first studied in the context of combinatory logic, was applied to Church's λ-calculus in [33] and did not require modification of the syntax. In that approach, given an untyped λ-term, the question of whether the term is typeable (or not) is answered by constructing (or failing to construct) a suitable justification. Such proofs come in the form of *typing derivations*, which when constructed also give a possible type for the term. The inference rules for constructing typing derivations are given below.

Definition 2.3.6 (Typing Derivations for the λ-calculus, [11]) *A typing derivation is a tree whose leaves are instances of the rule* (Ax) *and intermediate nodes are instances of the rules* (Abs) *and* (App).

$$\frac{}{\Gamma, x{:}A \vdash x{:}A}\,(Ax) \qquad \frac{\Gamma, x{:}A_1 \vdash M{:}A_2}{\Gamma \vdash \lambda x.M{:}A_1 \to A_2}\,(Abs) \qquad \frac{\Gamma \vdash M{:}A_1 \to A_2 \quad \Gamma \vdash N{:}A_1}{\Gamma \vdash (MN){:}A_2}\,(App)$$

1. *A* statement *is an expression of the form* $M{:}A$. *The* λ*-term* M *is called the subject and the type* A *is the predicate of the statement.*
2. *A* context Γ *is a set of statements with only distinct variables as subjects. We write* $\Gamma, x{:}A$ *to denote* $\Gamma \cup \{x{:}A\}$.

3. *We will write* $\Gamma \vdash M{:}A$ *if the statement is derivable. i.e., if there exists a derivation with that statement in the bottom line built using the three rules given.*

A proof that the term $(\lambda x.x)y$ is typeable is given below. The term is typeable with A (for all types A).

$$\dfrac{\dfrac{\dfrac{}{x{:}A \vdash x{:}A}\,(Ax)}{x{:}A \vdash (\lambda x.x){:}A \to A}\,(Abs) \qquad \dfrac{}{y{:}A \vdash y{:}A}\,(Ax)}{x{:}A, y{:}A \vdash ((\lambda x.x)y){:}A}\,(App)$$

While the type systems presented above can be commended for capturing only terms which satisfy the properties of mathematical functions, it is clear that many equally good functions are not captured—in particular all recursive functions are excluded. The terms of the simply-typed fragment of the λ-calculus are *strongly normalisable*, guaranteeing for any term that all reduction paths will reach the normal form within a finite number of steps. It follows intuitively that this fragment is no longer Turing-complete. There are several approaches to restoring Turing-completeness.

The interaction between a general recursor $\mathcal{R}$ and a program M is given by,

$$\mathcal{R}M \to M(\mathcal{R}M)$$

Analysing the rule, the recursor should be typed with $(A{\to}A){\to}A$, though there is no closed λ-term which has this type. There are, however, λ-terms that exhibit the reduction behaviour of $\mathcal{R}$, for example the combinator:

$$\mathcal{Y} \stackrel{\text{def}}{=} (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$$

However, such terms cannot be typed due to the (untypeable) self-applications.

Unfortunately, typeable recursion cannot be expressed in the simply-typed fragment of the λ-calculus, and more generally one cannot type non-terminating programs. One approach of getting round this is to enrich the language with a typed constant such as $\mathcal{Y}$, thus allowing the reduction behaviour suggested above for $\mathcal{Y}$ (as done for ML), but essentially treating the inner workings of the program $\mathcal{Y}$ as a black-box.

2.3.2 An Introduction to Control

As real-world programming languages were developed, languages were enriched with different commands which captured useful computational behaviour. In particular, a number of 'control' structures that reorganised the natural sequential flow of a program were designed. This section is intended to introduce the reader to ideas of control and continuations in programming languages. Given the prevalence of imperative programming languages in practice, we will introduce concepts in this setting before transferring notions to the functional paradigm.

An Imperative Description of Control

Procedural languages enjoy the benefits of encapsulation, where one is given the facility to group together some functionality for clarity and re-use. When first devised, computer memory was limited and costly making it an extremely attractive paradigm from a practical perspective.

Functional procedures share similarities with mathematical functions: they are passed inputs which they operate on and compute a value, which is then 'returned' to the caller of the function. Perhaps the key feature of the procedure call is the provisions it makes for resumption of program execution upon completion of its invocation. Each procedure call disturbs a program's natural *flow of control* from one instruction to the next sequentially listed instruction.

We will look at this process of invoking a procedure in more detail.

Consider the following excerpt of code, where points in the program have been labelled with numbers.

```
1  int add(int x, int y){
2      return x+y;
3  }
4
5  int divideTwo(int x){
6      return x/2;
7  }
8
9  int average(int x, int y){
10     return divideTwo(add(y,x));
11 }
```

Listing 2.1: Procedure Call

The function `average`, when invoked with a valid inputs, calculates the average of the two arguments, then returns that value.

The definition of each procedure leaves implicit the location where execution should resume following its call—it is simply written that the procedure will 'return' some value. This is of course necessary for the procedure's use and re-use at different points of the program.

A compiler will translate this code into the language of the machine. When the compiled program is run, a record of the currently executing instruction is stored in an *instruction pointer*. The instruction being pointed to is said to have *control*. For each procedure call, the generated compiled code will say where control should jump to and resume from by manipulating the instruction pointer. Constructs, such as procedure calls, that alter the instruction pointer's natural behaviour (from the current instruction to the next sequentially listed instruction) are called *control structures*.

During each transfer of control, additional information such as the values of variables may also require saving (and later restoring). We will introduce two concepts (*scopes* and *frames*) which are necessary to explain the reasons for this.

Scopes define the accessibility of program variables, i.e., where variable names may be used. They also serve to keep variable names in different parts of the program distinct from one another. When control enters a scope, the named variables may be associated with values. In the example above, each scope is indicated with a pair of curly braces. During the execution of a program, a frame is associated with each scope with the purpose of associating a value with each

variable in the scope.

A procedure defines a new scope, so when called during execution, a new frame is created to record information specific to that scope. This new frame is stacked on top of the current frame (which associates variables with values in the current scope). For procedures, the new frame records at least the following information: (i) the value associated with each parameter by the caller (ii) the return address of the procedure. We will look at an example to illustrate this.

Example 2.3.7 (Applicative-style Program Execution) *The invocation of the procedure* `average(2,4)` *proceeds as follows.*

1. *Control enters at line 9. Memory for a new frame F_1 will have been allocated, mapping the formal parameters* `x,y` *to the actual values* `2,4` *respectively.*
2. *Control is passed to line 10, where a call to* `add(4,2)` *is made.*
3. *Control passes to line 1.*
4. *The procedure* `add` *defines a new scope, so memory is allocated for a new frame F_2, which maps the variables* `x,y` *to* `4,2` *respectively, and records the fact that control must be returned to line 10 when the procedure completes.*
5. *The frame F_2 is added to the top of the stack frame, saving the old frame F_1 for possible later use.*
6. *Control passes to line 2. Using machine level instructions, the expression* `4+2` *is evaluated to* `6`*.*
7. *The return statement is evaluated, which places the resulting value at an agreed location which is accessible to the code at line 10 — e.g. in a machine register, some shared memory space, in F_1's frame, etc.*
8. *Control is passed back to line 10, and the topmost frame F_2(with $x \mapsto 2, y \mapsto 4$) associated with the* `add` *procedure is popped off the stack, releasing the allocated memory.*

The procedure continues with the call `divideTwo(6)` *in the same way, but for our purposes we have highlighted the key steps of procedure calling.*

The example above describes how control is restored to the function caller after the procedure call. The `return` instruction can be seen as a special kind of jump 'procedure' instruction with arguments. It has the effect of abandoning the current frame, restoring the frame of the procedure caller, then passing its parameter (the value of the procedure's computation) to some other program-point, where execution is resumed from. If we were to package up this information (the re-

sumption program-point and the restored frame), we would obtain what is referred to in the literature as a *continuation*.

Notice in this example, all information regarding frames and instruction points is implicit. A number of different methods exist which can be used to make these continuations explicit. Van Wijngaarden, amongst others [75], suggested introducing a special class of continuation objects to the language.

In fact, early compilers transformed such programs into an intermediatory style of language known as a *continuation passing style* (*CPS*), where such continuations were made explicit by representing them as special functions.

Programs in CPS do not return control to their caller—there is no return statement. Instead, every procedure is modified to take an extra parameter, namely a continuation. These are thought of as special procedure-like objects which when invoked (perhaps with a parameter), continue the execution of the program. They specify what should happen next, after the procedure body has been evaluated.

To exemplify this, we will reformulate the above example in a continuation passing style. There is however the 'small' problem of types. Procedures no longer return and therefore have no return type[9]. We therefore omit the return types from procedures, writing instead the procedure identifier `proc`. The second problem is the type of the continuation function which the result of the procedure should be passed to. Clearly this type can vary, depending on what type the procedure previously returned (and now instead passes to the continuation). We will cover types for continuations in Section 2.3.4. For now, we introduce an 'umbrella type' `continuation` to cover all the different cases. Objects of this type are procedures which when invoked will execute and not return control to the caller.

Example 2.3.8 (Continuation-style Program Execution) *This example describes how to transform the procedure call in Listing 2.1 into a continuation passing style. We can easily transform the procedures* `add` *and* `divideTwo` *as follows.*

[9] notice that this is different from having a void return type, which would specify that the procedure returns no value

```
proc add(int x, int y, continuation c){
    c(x+y);
}

proc divideTwo(int x, continuation c){
    c(x/2);
}
```

Transforming the procedure `average` *is slightly more difficult due to the composition of procedure calls. In CPS, we can no longer nest procedures in the usual way since values are not returned to the caller. However, with some effort, we can build a scenario where the result of one procedure is passed to another using a continuation. The steps to compute* `average(x,y,c)` *which passes its result to a continuation* `c` *would include the following steps.*

1. *The call to* `add(y,x,K)` *passes its result in some continuation 'function'* `K`*.*
2. *The call to* `divideTwo` *would use the argument passed to* `K`*, perform its computation (the divide two operation) then pass that result to the continuation function* `c`*, i.e., the call is of the shape* `divideTwo(result-passed-to-K,c)`

We can express this syntactically if we borrow some notation from the λ*-calculus.*

```
proc average(int x, int y, continuation c){
    add(y,x,λk.divideTwo(k,c));
}
```

The function K we spoke of was in fact λk.`divideTwo(`k`,c)`*. Now consider the procedure call* `average(2,4,c)`*.*

```
          average(2,4,c)
runs-to→  add(4,2,λk.divideTwo(k,c))
runs-to→  λk.divideTwo(k,c) (4+2)
runs-to→  λk.divideTwo(k,c) 6
runs-to→  divideTwo(6,c)
runs-to→  c(6/2)
runs-to→  c(3)
```

Notice that the transformation to a continuation style modifies the structure of the program. In the procedural style, `add` *is the inner function and* `divideTwo` *is the outer, while in the continuation style the composition has been reversed.*

The function calls in the continuation style are all tail-calls*, meaning there are no pending operations to be performed when a function is called. For instance, notice that the call to* `add` *from* `average` *in in the procedural style (Listing 2.1) is* not *a tail-call because there is a pending call to the* `divideTwo` *procedure before the* `average` *procedure can exit.*

A full CPS transform works on the *entire* program and does much more than described by the example. However, we will end our discussion of CPS transforms here since we have fulfilled our goal of demonstrating how one may use continuations to pass results from one program-point to another. We will now look at these features in the more formal setting of term-calculi.

Control in λ-calculi

Listing 2.1 could have been expressed in the following λ-calculus syntax enriched with natural numbers and arithmetic operations.

Example 2.3.9 (Translation of Listing 2.1 to (enriched) λ-calculus)

$$\begin{aligned} \mathit{add} &\equiv \lambda x.\lambda y.(x + y) \\ \mathit{divideTwo} &\equiv \lambda v.(v/2) \\ \mathit{average} &\equiv \lambda x.\lambda y.(\mathit{divideTwo}\ (\mathit{add}\ y\ x)) \end{aligned}$$

In λ-calculi there is no 'return' keyword: the computed value of a function application is passed to the enclosing term called the *evaluation context*.

Here we introduce the syntax '$[\,]$' to represent an evaluation context, which can be thought of as a term with a single 'hole'. If E is an evaluation context, then $E[M]$ denotes the term that results by plugging the hole of E with the term M. In the function *average*, the evaluation context of the $(\mathit{add}\ y\ x)$ call is the term that surrounds it: $\lambda x.\lambda y.\lambda z.(\mathit{divideTwo}\ [\,])$, and $[\,]$ represents a hole where the term use to sit. Once $(\mathit{add}\ y\ x)$ is evaluated, its value (the result of $x + y$) plugs the hole of the evaluation context. Notice that, in general, the evaluation context is the part of the program that specifies what should happen after the result has been returned to the hole—it is the *continuation* of the program.

In the pure λ-calculus, notions of control are left implicit—it is understood that results of computations will always be returned to their enclosing contexts. For this reason, it is fairly well accepted that the λ-calculus is not well-suited to the

study of control. Those wishing to formally study control features have typically taken the following approaches.

(Use of CPS transformations) : This method of transforming an entire source program results in a λ-calculus program where (i) all return locations in the original program are made explicit, (ii) all intermediate values of a computation are named, (iii) an order of evaluation is fixed and (iv) the flow of control or (c.f. the call-stack) is 'inverted'. Since target programs are not at all humanly understandable, this approach of studying control will not be discussed further in this thesis.

Extension of existing languages : Well studied calculi are extended with constants corresponding to specific notions of control (e.g. $\lambda_{\mathcal{C}}$).

Design of new term-calculi : Languages are designed from formal logical foundations with explicit syntax for control structures (e.g. first class continuations and continuation delimiters). This approach generally allows computational behaviours to be studied at varying (and often finer) degrees of granularity.

The second approach was taken by Felleisen *et al.* [37] who studied Landin's ISWIM language and developed the $\lambda_{\mathcal{C}}$-calculus.

The $\lambda_{\mathcal{C}}$-calculus has three syntactic categories of expressions: terms, values and evaluation contexts. Terms are the usual λ-calculus terms plus three constants $\mathcal{A}, \mathcal{K}$ and $\mathcal{C}$ which describe some specific control behaviours. The 'value' category is introduced to restrict the applicability of the certain reduction rules. An explicit representation of an evaluation context (or 'continuation') is also given. The formal definition is given below.

Definition 2.3.10 ($\lambda_{\mathcal{C}}$-syntax) *Where $x, y, z \ldots$ range over an infinite set of variables, M, N range over terms, V ranges over values and E ranges over evaluation contexts, the language of the $\lambda_{\mathcal{C}}$-calculus is defined by the following grammar.*

$$\begin{array}{lll} M, N & ::= & V \mid MN \mid \mathcal{A}M \mid \mathcal{K}M \mid \mathcal{C}M \\ V & ::= & x \mid \lambda x.M \\ E & ::= & [\,] \mid EM \mid VE \end{array}$$

Notice that this calculus has explicit syntax for evaluation contexts. The language $\lambda_{\mathcal{C}}$ has been especially designed so that any non-value $\lambda_{\mathcal{C}}$-term $E[R]$ can be uniquely decomposed into a redex R and an evaluation context $E[\,]$, implying

a fixed evaluation strategy. For example, the $\lambda_{\mathcal{C}}$ term $((\lambda x.\lambda y.x)v)((\lambda w.w)z)$, despite having several redexes, can only be decomposed (according to the rules of the grammar) as follows.

$$\begin{aligned} E[\,] &= ([\,])((\lambda w.w)z) \\ R &= (\lambda x.\lambda y.x)v \end{aligned}$$

Reduction in the $\lambda_{\mathcal{C}}$-calculus extends the usual notion of β-reduction with reduction rules that give the behaviour of the constants $\mathcal{A}, \mathcal{K}$ and $\mathcal{C}$:

$$\begin{aligned} (\mathit{EvalAbort}): &\quad E[\mathcal{A}M] \rightarrow M \\ (\mathit{EvalControl}): &\quad E[\mathcal{C}M] \rightarrow M(\lambda x.\mathcal{A}E[x]) \\ (\mathit{EvalCallCC}): &\quad E[\mathcal{K}M] \rightarrow E[M(\lambda x.\mathcal{A}E[x])] \end{aligned}$$

Notice that each use of a control operator manipulates its evaluation context in a specific way.

- An invocation of the 'abort' operator, $\mathcal{A}$, throws away the current evaluation context and continues the evaluation of its argument M.
- The 'control' operator $\mathcal{C}$ is slightly more complicated. First, it creates a special function, $\lambda x.\mathcal{A}E[x]$, whose behaviour is comparable to the 'throws' operator used in exception handlers: if applied to an argument (i.e., if 'invoked' with a parameter) it will abandon any computation and immediately return that argument to the context E. The term M is applied to this special function. Consider the case when M is of the shape $\lambda y.N$. The reduct is $(\lambda y.N)(\lambda x.\mathcal{A}E[x])$—notice the 'throws' function is passed to the body of the abstraction via the formal parameter y. The function body N may then invoke the 'throws' function, perhaps signalling exceptional conditions, by applying y to some value.
- The 'call/cc' operator $\mathcal{K}$ is similar to $\mathcal{C}$, except a value is always returned to the context of the call, whether or not the 'throws' function is used. This prompts the question why $\mathcal{A}$ is applied to the reified context in the throws function: would it not have been simpler to reify the current evaluation context as $(\lambda x.E[x])$. The essential feature to notice here is that the abort operator abandons any other computations which may occur in the evaluation of M. Practically, we can think of $\mathcal{K}$ as a shortcut operator for lazy evaluation. Imagine its use in a list searching procedure: once the sought element has been found, the rest of the list need not be traversed, so the element can be 'thrown' back to the evaluation context.

These control operators have a variety of practical uses, and their definition in a formal setting allows their properties to be cleanly studied. They do however perform fairly coarse operations: the *entire context* of the call is captured in a *single step*. Felleisen observed that capturing and reifying the whole context is expensive and often unnecessary, and in [38], he introduced a notion of delimited contexts. These delimiters were used to 'section off' the portion of the context to be reified. Several alternative calculi have been proposed and studied which feature context delimiters. However, the origins of these calculi are quite different. We will return to context delimiters in Sections 2.3.4 and 2.3.5, but first we will discuss a key motivation behind the study of this computational feature.

2.3.3 Curry-Howard Correspondences

The typing rules for the λ-calculus (see Definition 2.3.6) bear a strong resemblance to Gentzen's Natural Deduction style of inference rules for the logical implication connective (see Section 2.2). The relationship between these two formalisms was studied by Howard [49] in 1969, though a relationship between logic and computing had been remarked upon much earlier by Curry and Feys [33] in 1958. De Bruijn [22] also noticed this relation when he used λ-calculus terms to represent of proof objects in his AUTOMATH tool.

The 'Curry-Howard Isomorphism' describes the strong relationship between the propositions of minimal implicative logic and the simply-typed λ-calculus, namely that (i) propositions correspond to types (ii) proofs of propositions can be represented with λ-terms and (iii) proof normalisation corresponds to β-reduction.

Since the initial discovery, various researchers have studied this relation in different contexts seeking to extract other correspondences between various logics and models of computation. For a long time, it was believed that such a correspondence only existed between constructive logics and computational calculi. However, Griffin [43] uncovered such a relationship between Classical Logic and the control operators in Felleisen's $\lambda_{\mathcal{C}}$-calculus. He was able to assign types corresponding to classical tautologies to the operators $\mathcal{C}$ and $\mathcal{K}$.

By typing the 'plugging of a context's hole with a term' with a form of the logical cut rule, and observing the reduction behaviour of Felleisen's $\lambda_{\mathcal{C}}$ control operators, Griffin was able to assign the following classical types, corresponding to the classical tautologies Double-Negation Law (DN) and Peirce's Law (PL), to $\mathcal{C}$ and $\mathcal{K}$ respectively.

Definition 2.3.11 (Griffin's Type Assignment, [43]) *The typing rules of the λ-calculus (Definition 2.3.5) were augmented with the following typing rules for the extra syntactic constructs.*

$$\frac{\Gamma \vdash M{:}A \qquad \Gamma, [\,]{:}A \vdash E[\,]{:}B}{\Gamma \vdash E[M]{:}B}\,(EvalCtx) \qquad \frac{\Gamma \vdash M{:}\bot}{\Gamma \vdash (\mathcal{A}M){:}A}\,(\bot E)$$

$$\frac{\Gamma \vdash M{:}(A\to\bot)\to\bot}{\Gamma \vdash (\mathcal{C}M){:}A}\,(DN) \qquad \frac{\Gamma \vdash M{:}(A\to B)\to A}{\Gamma \vdash (\mathcal{K}M){:}A}\,(PL)$$

Note that negation is not explicitly represented in the language of types. Instead, it is defined in terms of implication and bottom, i.e. $\neg A \stackrel{def}{=} A\to\bot$.

Remark 2.3.12 (Abort Operator) *Observing the reduction behaviour of $\mathcal{A}$, M is actually typeable with any type B, leading to the more general type $(B\to A)$ for $\mathcal{A}$. However, falsity is used since Griffin wants to create a correspondence with a logic and therefore requires logical consistency.*

Remark 2.3.13 (Control Operator) *Griffin's type assignments were motivated by classical logic. The control operator $\mathcal{C}$ could have been assigned the more general type of $((A\to B)\to\bot)\to A$. Whether or not $\mathcal{C}$ is the 'best' inhabitant for $\neg\neg A\to A$ has recently been a subject of debate. Summers [83] criticises Herbelin's criticism of de Groote [44] and Ong and Stewart [67], who all seek Curry-Howard correspondences for Classical Logic and study the control behaviour of terms corresponding to the Double-Negation law. Summers argues that in each of these cases, the terms extracted correspond better to the behaviour of Felleisen's control operator $\mathcal{F}$ than to the behaviour of $\mathcal{C}$.*

We remark that the original Curry-Howard correspondence was discovered between two formalisms that were well studied and very well understood. In contrast, the many Curry-Howard correspondences that have been proposed for Classical Logic have employed non-standard presentations of the logic (which have not been well studied or defined ad hoc) and/or some computational behaviour that either is not well understood or does not fit to the logic very well. This is seen in Griffin's system who has to make several *design choices* (as remarked upon above) to obtain the correspondence; even then, the system is not perfect since subject reduction does not hold in general.[10] We will study some of

[10]Several fixes have been suggested for this: Griffin himself proposes wrapping up all programs in special contexts, while Ariola and Herbelin [2] propose adding a special 'top-level' constant to the language. However, both of these changes does not make the correspondence any more natural.

the proposed Curry-Howard correspondences for Classical Logic in the sections to come.

The reason for the difficulty, in our opinion, is the result of a number of factors. First, the presentation of a classical logic in the Natural Deduction calculus involves the addition of a single rule[11]. This breaks the pattern of 'introduction' and 'elimination' rule pairs, that are traditionally understood (in a computational setting) as corresponding to term 'construction' and term 'destruction'. If we argue that classical logics are best presented in sequent calculi, then the non-confluence of cut-elimination becomes a key problem when seeking to interpret computational behaviour. Also, if we allow the sequent calculus' right introduction rules to be understood in the same way as natural deduction style 'introduction' rules, what should the left *introduction* rules correspond to computationally?

Some of these problems have already been addressed (and in spite of the difficulties, progress in the area has been constant) following Griffin's original intuitions: that classical propositions correspond to types for terms that represent or manipulate continuations of programs. In reviewing the various works seeking Curry-Howard style correspondences between Classical Logic and models of computation, we have noticed the approaches taken can be fitted into one of three distinct camps:

1. Assignments of classical propositions as types for computational operations. (e.g. Griffin's typing of $\lambda_{\mathcal{C}}$).
2. Extraction of term calculi from formal logical systems that have been carefully designed with properties of computational behaviour in mind (e.g. Parigot's $\lambda\mu$ [68], Ong and Stewart's $\lambda\mu_{\text{CBV}}$ [67], Herbelin's $\bar{\lambda}$ and $\bar{\lambda}\mu$ [46], Curien and Herbelin's $\overline{\lambda}\mu\tilde{\mu}$ [32], Wadler's Dual Calculus [91], Kesner's Typed Pattern Calculus [56])
3. Investigations into term calculi resulting from pure formal logical systems (e.g. Urban's calculus, Lengrand's $\lambda\xi$, van Bakel and Lescanne's $\mathcal{X}$ [9], Lescanne and Žunič's ${}^{*}\mathcal{X}, {}^{d}\mathcal{X}, {}^{\copyright}\mathcal{X}$ [63, 90], Summers' $\nu\lambda\mu$ [83])

The second and third camps are similar, in that they both try to fit a computational term calculus to a formal logical system for Classical Logic, yet there is an important and notable distinction separating them.

Researchers in the second camp hold a pre-conceived notion of how computation

[11] In fact, there is a choice of which rule to add which can yield classical logics of different strengths. Ariola *et al.* [3] identify these rules as Peirce's Law, the Double Negation Elimination or the Law of Excluded middle.

may (and may not) behave *before* the design of their 'Curry-Howard' systems. There is a strong view that confluence is an essential feature of the reduction mechanism of any computational term calculus. Researchers holding this view have therefore sought to restrict the normalisation mechanism of proofs in the design of their 'Curry-Howard' logic to also be confluent. Finding suitable proof rule inhabitants that express some computational behaviour is also non-trivial, and often leads to modifications of the underlying logical inference rules.

The third camp of researchers tries not to make any changes to the logical system influenced by computational notions, and instead attempt to extract and study the computational content that is naturally apparent in the system. Since there is no pre-conceived notion of computation, the resulting term-calculus syntax is often a very straightforward proof annotation. It is also worth noting that the resulting term calculus need not be 'well-behaved' in the conventional sense. These points have been criticised by members of the second camp and of course are points which need to be justified: especially the presence of non-confluence.

Nevertheless, the research in this thesis approaches the design of Curry-Howard calculi with the mind-set of the third camp. Many of the calculi studied here are indeed both non-confluent and non-deterministic. However, as we will see in Chapter 3 where we study properties of the $\mathcal{X}$-calculus, interesting confluent subsystems can arise from highly non-confluent term calculi. Additionally, nice symmetries present in the underlying logic are automatically preserved.

The third camp of thought is fairly recent in comparison to the other approaches, but is not independent of them. The $\mathcal{X}$-calculus is a result of lines of work taken by Herbelin [32] and Urban [86], and in fact much insight we have gained into the computational understanding of $\mathcal{X}$ can be attributed to the works of the second camp. We will therefore review the key insights that led to the development of $\mathcal{X}$ in the following subsections.

2.3.4 On Parigot's $\lambda\mu$ Calculus

Parigot criticises Griffin's system as being "not satisfactory from a logical point of view" and in [68] presents his own typed computational calculus, the $\lambda\mu$-calculus, which holds a (restricted) Curry-Howard style correspondence with Classical Logic. Parigot's departure point [69] is interesting since he argues that "neither natural deduction nor sequent calculus provide a suitable cut-elimination" mechanism of computation, for the reasons that confluence and strong normal-

isation are not effectively apparent. He introduces a new logical framework to study Classical Logic which he calls 'free deduction', designed to exhibit the desired computational properties.

The terms of his $\lambda\mu$-calculus inhabit classical proofs formulated in free deduction. We present the propositional fragment of his calculus below, followed by an explanation of its operations.

Definition 2.3.14 ($\lambda\mu$-calculus [68]) *The language of the λ-calculus (Def. 2.3.1) is extended with μ-variables $\alpha, \beta, \ldots$ that range over the infinite set of* names. *$\lambda\mu$-terms are an extension on the set of λ-terms and are constructed with the following grammar.*

$$\begin{array}{lccccccc} M, N ::= & x & | & (\lambda x.M) & | & (MN) & | (\mu\alpha.M) & | \ ([\alpha]M) \\ & \textit{variable} & & \textit{abstraction} & & \textit{application} & \textit{activate} & \textit{passivate} \end{array}$$

We will omit unnecessary brackets.

The symbol μ used in the activate term identifies the adjacent name as a formal parameter: a name bound over the term M. The name α is free in the passivate term $[\alpha]M$.

The set of types from Definition 2.3.5 is extended with a constant symbol $\bot$. Typing derivations for the $\lambda\mu$-calculus are constructed with the following rules, inspired by the inference rules of free deduction.

Definition 2.3.15 (Typing Assignment Rules for $\lambda\mu$-calculus, [68]) *The symbols Γ and Δ represent sets of types labelled with variables and names respectively.*

- *Logical Rules*

$$\frac{}{\Gamma, x{:}A \vdash_{\lambda\mu} x : A \mid \Delta}\,(Ax) \qquad \frac{\Gamma, x{:}A_1 \vdash_{\lambda\mu} M : A_2 \mid \Delta}{\Gamma \vdash_{\lambda\mu} \lambda x.M : A_1 \to A_2 \mid \Delta}\,(Abs)$$

$$\frac{\Gamma \vdash_{\lambda\mu} M : A_1 \to A_2 \mid \Delta \qquad \Gamma \vdash_{\lambda\mu} N : A_1 \mid \Delta}{\Gamma \vdash_{\lambda\mu} MN : A_2 \mid \Delta}\,(App)$$

- *Naming Rules*

$$\frac{\Gamma \vdash_{\lambda\mu} M : \bot \mid \alpha{:}A, \Delta}{\Gamma \vdash_{\lambda\mu} \mu\alpha.M : A \mid \Delta}\,(Activate) \qquad \frac{\Gamma \vdash_{\lambda\mu} M : A \mid \Delta}{\Gamma \vdash_{\lambda\mu} [\alpha]M : \bot \mid \alpha{:}A, \Delta}\,(Passivate)$$

The right-hand side of the judgements are of an usual shape: $M{:}A \mid \Delta$. The position to the left of the vertical bar is known as the *stoup*, and allows one to distinguish a particular element from all the elements to the right of the turnstile, i.e., it focuses a particular conclusion in a proof.[12] The (*Activate*) and (*Passivate*) rules are structural rules (from the point of view of the logic) and respectively move an element in or out of the stoup position. In this way, the formation of a $\lambda\mu$-term is tightly controlled, ensuring a unique mapping between proofs and terms. We remark that the (*Activate*) rule plays a second role, namely that it captures a modified form of the Proof by Contradiction Law (PC). Allowing for some leniency on notation, the (PC) rule could be formulated as,

$$\frac{\neg\Delta, \Gamma, \neg A \vdash \bot}{\neg\Delta, \Gamma \vdash A}\,(PC)$$

Parigot defines a computational mechanism on these new constructs, known as *structural substitution*. Whereas in the λ-calculus, one is only permitted to apply functions to arguments when they sit at the same level of the term syntax tree, the additional naming constructs pull contexts to named locations, building applications with subterms at arbitrary depths.

Definition 2.3.16 (Reductions in $\lambda\mu$) *The reduction rules for the $\lambda\mu$-calculus, in addition the usual β-reduction rule, are defined as follows:*

$$\begin{array}{lll}
(\mu\text{-}structural_R): & (\mu\alpha.M)N \to \mu\beta.M\{[\beta](P\ N)/[\alpha]P\} & \beta\ fresh \\
(\mu\text{-}renaming): & [\beta]\mu\alpha.M \to M\{\beta/\alpha\} & \\
(\mu\eta): & \mu\alpha.[\alpha]M \to M & \leftarrow\ \alpha \notin M
\end{array}$$

There is in fact a second important structural rule, shown below.

$$(\mu\text{-}structural_L): \quad N(\mu\alpha.M) \to \mu\beta.M\{[\beta](N\ P)/[\alpha]P\} \quad \beta\ fresh$$

We assume that this rule was omitted from the design of the system to avoid the critical pairs $(\mu\alpha.M)(\mu\alpha.N)$ and $(\lambda x.M)(\mu\alpha.N)$ which would result in a non-confluent system. The omission also leads to a weaker normalisation procedure of proofs in the underlying logic.

The rule (μ-*structural*$_R$) should be understood as follows. There are subterms occurring in M that have been named using the passivate construct; these named

[12] This construction maintains the simplicity of sets, but allows one to distinguish a particular element without having to resort to the full overhead of exchange-rules that would be needed if sequences were used.

subterms are of the shape $[\alpha]P$. The activate term binding over these named subterms represents a *handle* to those named subterms, that is, a way for the subterms to be accessed at some outer-level. The application of an argument N to the activate term, $\mu\alpha.M$, represents the action of applying the argument N directly to those named subterms appearing in M, i.e., the application PN for every $[\alpha]P$. The resulting subterm is given a new name, hence $[\beta](PN)$. The outer-level handle to the terms at these named positions is not lost, hence the re-introduction of the activation term binding over those subterms named with $[\beta]$.

Intuitively, we can think of the rule in (μ-*structural*$_R$) as an application of a named subterm P (that are ideally functions) in M to the provided argument N. The (μ-*structural*$_L$) represents the symmetric situation where the named subterms P in M are ideally values, waiting for a function N to be applied to them. But notice that more than one argument or function can be used in such an application since the outermost μ-binder persists in the structural reduction rules. For this reason, an understanding of the renaming and eta rules is equally important. Consider the symmetrical cases,

$$(((\mu\alpha.M)N_1)N_2)\ \ldots \tag{2.1}$$

$$\ldots(Q_2(Q_1(\mu\alpha.M))) \tag{2.2}$$

The term of (2.1) is able to consume and apply, *to* the named subterms of M, an infinite number of arguments N_i as long as an argument N_i is not of the shape $[\alpha]N_i'$. Similarly, the term of (2.2) is able to consume and apply an infinite number of subterms Q_i to the named subterms of M, until a term of shape $[\beta](Q_i(\mu\alpha.M))$ is encountered. Since the continuations of $(\mu\alpha.M)N_1$ (or $Q_1(\mu\alpha.M)$) are the enclosing applications (loosely speaking $([\,]N_i\ldots)$ or $(\ldots Q_i[\,])$), the passivate terms can be seen as having a second purpose: they are 'delimiters' for contexts specifying how much of the entire context consumed during the structural substitutions.

With this understanding of the reduction behaviour, the $\lambda\mu$-calculus can be used to express the control operators constants $\mathcal{A}$, $\mathcal{C}$ and $\mathcal{K}$ as follows.

$$\begin{aligned}
\mathcal{A}M &\stackrel{\text{def}}{=} \mu\alpha.M && \leftarrow\ \alpha \notin fv(M)\\
\mathcal{C}M &\stackrel{\text{def}}{=} \mu\alpha.M(\lambda z.[\alpha]z)\\
\mathcal{K}M &\stackrel{\text{def}}{=} \mu\alpha.[\alpha](M(\lambda x.\mu\beta.[\alpha]x))
\end{aligned}$$

Since there are explicit term representations of the constants, the $\lambda\mu$-terms may take several rewrite steps (as opposed to a single step), reflecting a finer grained reduction mechanism.

We also notice that the activate and passivate terms have two separate and quite disjoint responsibilities: they are used to delimit contexts and *also* provide a simple naming mechanism for subterms. This joint responsibility is a direct result of the merging of the *proof by contradiction* law (PC) with the 'structural rules' that move formulas to and from the stoup position of the derivations in Parigot's formalism. We argue that the structural substitution rules are solely due to the normalisation of the underlying (PC) rule, while the 'context delimiting' feature is due to the portion of the rules concerned with the stoup. This is evident when we consider the origins of Parigot's structural substitution mechanism.

In [71, pp. 39-40], where Prawitz is concerned with normalising instances of the proof by contradiction law (he considers a proof with applications of the rule to be 'normal' if such applications are at the leaves of the derivation). He describes a proof transformation which builds proofs where the (PC) rule is applied only to atomic formulas. This transformation essentially describes the same process as Parigot's structural substitution mechanism, except Parigot has (necessarily) combined the instance of the (PC) rule (corresponding to the activate term) with the elimination rule for logical implication so his reduction mechanism will work in an untyped setting as well. Notice that the left-hand side of the structural reduction rules describe the cases of *function* application when an activate term is in the 'function' or 'argument' positions.

There is no generic method for normalising an instance of (PC) rule; its means of normalisation depends on the specific logical connective it is applied to. Prawitz defines the normalising proof transformation individually for the logical connectives: implication, conjunction and disjunction. This transfers over to calculi built in the style of $\lambda\mu$, where structural reduction rules (that combine the activate term with the term inhabiting a connective's inference rules) will be needed for each logical connective in the calculus.

Curien and Herbelin make an important step in [32] and are able to separate some of these concerns, essentially by moving to a sequent calculus setting. We will study their $\overline{\lambda}\mu\tilde{\mu}$-calculus in the next section.

2.3.5 Curien and Herbelin's $\overline{\lambda}\mu\tilde{\mu}$

In [32], Curien and Herbelin develop a term calculus called $\overline{\lambda}\mu\tilde{\mu}$ which holds a (restricted) Curry-Howard correspondence with the sequent calculus $\mathrm{LK}_{\mu\tilde{\mu}}$: a variant of Gentzen's sequent calculus for Classical Logic, LK. The calculus natu-

rally exhibits some of the symmetries of computation, namely the ideas of input versus output, program versus context and call-by-name versus call-by-value reduction.

A key feature of this calculus is that it is derived from a sequent calculus, as opposed to a natural deduction calculus. In [1], Ariola *et al.* noticed an interesting feature when basing notions of computation on sequent calculi: all procedure calls are naturally tail-calls. Such a transformation would usually require a full CPS transformation, at the same time making all continuations explicit. However, programs in the $\overline{\lambda}\mu\tilde{\mu}$-calculus are already in a continuation style.

We will present this calculus first in its purest form, which Herbelin calls the $\mu\tilde{\mu}$-subsystem. It has no logical connectives and is still capable of expressing some of the general ideas of computation.

Definition 2.3.17 ($\mu\tilde{\mu}$-syntax) *The language of the $\mu\tilde{\mu}$-subsystem is defined by the following syntax, where $x, y, z, \ldots$ range over the infinite set term variables, and $\alpha, \beta, \gamma, \ldots$ range over the infinite set of evaluation context variables (also called* names*).*

$$\begin{array}{lll} \textit{Commands} & c & ::= \langle v \parallel e \rangle \\ \textit{Terms} & v & ::= \mu\alpha.c \mid x \\ \textit{Evaluation Contexts} & e & ::= \tilde{\mu}x.c \mid \alpha \end{array}$$

The term constructors of this language inhabit sequent calculus style inference rules for the cut, the axiom and the structural rules (which shift a formula scheme in/out of the stoup position). The type assignment rules for the calculus are presented below.

Definition 2.3.18 (Typing rules for the $\mu\tilde{\mu}$-subsystem)

$$\frac{\Gamma \vdash_{\overline{\lambda}\mu\tilde{\mu}} v : A \mid \Delta \qquad \Gamma \mid e : A \vdash_{\overline{\lambda}\mu\tilde{\mu}} \Delta}{\langle v \parallel e \rangle : \Gamma \vdash_{\overline{\lambda}\mu\tilde{\mu}} \Delta}\,(Cut)$$

$$\frac{}{\Gamma, \alpha : A \vdash_{\overline{\lambda}\mu\tilde{\mu}} \alpha : A \mid \Delta}\,(Ax_R) \qquad \frac{}{\Gamma \mid \alpha : A \vdash_{\overline{\lambda}\mu\tilde{\mu}} \alpha : A, \Delta}\,(Ax_L)$$

$$\frac{c : \Gamma \vdash_{\overline{\lambda}\mu\tilde{\mu}} \alpha : A, \Delta}{\Gamma \vdash_{\overline{\lambda}\mu\tilde{\mu}} \mu\alpha.c : A \mid \Delta}\,(\mu) \qquad \frac{c : \Gamma, x : A \vdash_{\overline{\lambda}\mu\tilde{\mu}} \Delta}{\Gamma \mid \tilde{\mu}x.c : A \vdash_{\overline{\lambda}\mu\tilde{\mu}} \Delta}\,(\tilde{\mu})$$

Computational notions are based on the interaction between an evaluation context e and a term v using a construct called a *command*. The command $\langle v \parallel e \rangle$ is

typed with the cut rule, and has been compared to the applicative style construct $E[V]$, which is also typed with the cut.

Every program has at its leaves either a term variable x or an evaluation context variable α.

The term $\mu\alpha.c$ represents a computation (a 'command') c which when evaluated will pass its result to the evaluation context α. Symmetrically, the evaluation context $\tilde{\mu}x.c$ is a command c that is waiting for a some value to be passed to it via x. The reduction behaviour of these expressions is formalised by the following reduction rules.

Definition 2.3.19 ($\mu\tilde{\mu}$ Reduction Rules) *The reduction rules for the $\mu\tilde{\mu}$-subsystem given below.*

$$(\mu)\colon \langle \mu\alpha.c \parallel e \rangle \rightarrow c\{e/\alpha\}$$
$$(\tilde{\mu})\colon \langle v \parallel \tilde{\mu}x.c \rangle \rightarrow c\{v/x\}$$

As can be seen by the rules, the exact behaviour of a command depends on the shape of its component term and evaluation context. In the (μ)-rule, e is specified as the continuation of the subterms in c that output on α. In the $(\tilde{\mu})$-rule, a term v is passed to c which was waiting for an input via x.

The reduction relation is non-deterministic and non-confluent due to the unjoinable critical pair $\langle \mu\alpha.c_1 \parallel \tilde{\mu}x.c_2 \rangle$. Curien and Herbelin show that, by always preferring a particular rule during a reduction, a confluent reduction subsystem is obtained. In fact, they show that preferring the (μ)-rule leads to a call-by-name subsystem while preferring $(\tilde{\mu})$ leads to call-by-value. We will discuss reduction subsystems in more detail in Section 2.3.7.

In [48], Herbelin showed that the $\mu\tilde{\mu}$-subsystem can be extended with a variety of logical connectives, remarking that Wadler's Dual Calculus is in fact a variant of $\overline{\lambda}\mu\tilde{\mu}$ that simply employs different connectives (i.e., it is an extension of the $\mu\tilde{\mu}$-subsystem with the logical connectives for conjunction, disjunction and negation). We will extend the $\mu\tilde{\mu}$-subsystem with logical implication, to obtain $\overline{\lambda}\mu\tilde{\mu}$.

Definition 2.3.20 ($\overline{\lambda}\mu\tilde{\mu}$-syntax) *The language of the $\mu\tilde{\mu}$-subsystem given in Definition 2.3.17 is extended with the following constructors.*

$$\begin{array}{lll} \textit{Terms} & v & ::= \ldots \mid \lambda x.v \\ \textit{Evaluation Contexts} & e & ::= \ldots \mid v \cdot e \end{array}$$

The term $\lambda x.v$ can be thought of as the usual function abstraction. The evaluation context $v \cdot e$ is a list of arguments with the term v at its head. Note that evaluation contexts associate to the left, i.e., $v_1 \cdot v_2 \cdot v_3 \cdot \ldots \cdot e = (((v_1 \cdot v_2) \cdot v_3) \cdot \ldots \cdot e)$.

Definition 2.3.21 (Typing rules for $\overline{\lambda}\mu\tilde{\mu}$-calculus) *The constructs introduced in Definition 2.3.20 can be typed with the following rules.*

$$\frac{\Gamma, x : A \vdash_{\overline{\lambda}\mu\tilde{\mu}} t : B \mid \Delta}{\Gamma \vdash_{\overline{\lambda}\mu\tilde{\mu}} \lambda x.t : A \to B \mid \Delta}\,(\to_R) \qquad \frac{\Gamma \vdash_{\overline{\lambda}\mu\tilde{\mu}} v : A \mid \Delta \quad \Gamma \mid e : B \vdash_{\overline{\lambda}\mu\tilde{\mu}} \Delta}{\Gamma \mid v \cdot e : A \to B \vdash_{\overline{\lambda}\mu\tilde{\mu}} \Delta}\,(\to_L)$$

Extensions of the $\mu\tilde{\mu}$-subsystem with logical connectives follow the pattern that right-introduction rules build terms ('expressions that are waiting for continuations to send its output') while the left-introduction rules build evaluation contexts ('expressions that wait for inputs'). The interaction between the functional term and evaluation context constructs is defined below.

Definition 2.3.22 (Reduction rules for $\overline{\lambda}\mu\tilde{\mu}$) *The reduction rules from Definition 2.3.19 are augmented with the following rule.*

$$(\to) : \ \langle \lambda x.v \parallel v' \cdot e \rangle \to \langle v' \parallel \tilde{\mu}x.\langle v \parallel e \rangle \rangle$$

Unlike the β-rule that deals with functions in the λ-calculus, the reduction rule above *translates* (in one step) the interaction between the function and argument list into a series of interactions expressed only in the language of terms and evaluation contexts. The left-hand side reads: "the function $\lambda x.v$ is applied to a list of functional arguments $v \cdot e$". The right-hand side reads "$\tilde{\mu}x.\langle v \parallel e \rangle$ is a term waiting for an input to be passed to it via x. The input passed to x is v'.

Example 2.3.23 (A Reduction in $\overline{\lambda}\mu\tilde{\mu}$) *Using a standard natural deduction to sequent calculus transformation given in [32] (the translation $^<$), the program of Example 2.3.9 expressed in $\overline{\lambda}\mu\tilde{\mu}$ is listed below.*

$$\begin{aligned} add^< &= \lambda x.\lambda y.(x + y) \\ divideTwo^< &= \lambda v.(v/2) \\ average^< &\to \lambda x.\lambda y.\mu\delta.\langle divideTwo^< \parallel (\mu\beta.\langle add^< \parallel y \cdot x \cdot \beta \rangle) \cdot \delta \rangle \end{aligned}$$

In the function $average^<$, notice that once $add^<$ has consumed the values in the argument list $y \cdot x \cdot \beta$, it will return its result to β. The $divideTwo^<$ term will consume the handle to β, and use the underlying value in its computation, finally sending its result to δ.

Interestingly, an intuitively correct final result cannot be reached without adding the following η-rule (which is not part of $\overline{\lambda}\mu\tilde{\mu}$, but is part of $\lambda\mu$).

$$\mu\alpha.\langle t \parallel \alpha\rangle \quad \rightarrow_\eta \quad t \quad \leftarrow \ \alpha \notin fv(t)$$

Let us consider the interaction of the average$^<$ term with the functional context $2 \cdot 4 \cdot \alpha$, where α is a continuation representing 'where the final result of the entire computation should be sent'.

$$\begin{array}{ll}
 & \mu\alpha.\langle average^< \parallel 2 \cdot 4 \cdot \alpha\rangle \\
= & \mu\alpha.\langle \lambda x.\lambda y.\mu\delta.\langle divideTwo^< \parallel (\mu\beta.\langle add^< \parallel y \cdot x \cdot \beta\rangle) \cdot \delta\rangle \parallel 2 \cdot 4 \cdot \alpha\rangle \\
\rightarrow^* & \mu\alpha.\langle \mu\delta.\langle divideTwo^< \parallel (\mu\beta.\langle add^< \parallel 4 \cdot 2 \cdot \beta\rangle) \cdot \delta\rangle \parallel \alpha\rangle \\
\rightarrow^* & \mu\alpha.\langle \mu\delta.\langle divideTwo^< \parallel (\mu\beta.\langle (4+2) \parallel \beta\rangle) \cdot \delta\rangle \parallel \alpha\rangle \\
= & \mu\alpha.\langle \mu\delta.\langle divideTwo^< \parallel (\mu\beta.\langle 6 \parallel \beta\rangle) \cdot \delta\rangle \parallel \alpha\rangle \\
\rightarrow_\eta & \mu\alpha.\langle \mu\delta.\langle divideTwo^< \parallel 6 \cdot \delta\rangle \parallel \alpha\rangle \\
\rightarrow^* & \mu\alpha.\langle \mu\delta.\langle (6/2) \parallel \delta\rangle \parallel \alpha\rangle \\
= & \mu\alpha.\langle \mu\delta.\langle 3 \parallel \delta\rangle \parallel \alpha\rangle \\
\rightarrow_\eta & \mu\alpha.\langle 3 \parallel \alpha\rangle
\end{array}$$

The reduction steps describe computation at a greater level of verbosity, and as discussed in [1], the steps are closer to those performed by a stack based abstract machine.

To conclude, we would like to compare the reduction features of the $\mu\tilde{\mu}$-subsystem with those of calculi built in the style of Parigot's $\lambda\mu$. Recall that the reduction rules of Parigot's μ operators combine two aspects: (i) they copy contexts upto any delimeters and (ii) they perform a structural substitution.

In $\overline{\lambda}\mu\tilde{\mu}$, the delimiting of contexts using continuation variables is also present. A functional context such as $(v_1 \cdot v_2 \cdot v_3 \ldots \cdot \alpha)$ is a list of terms with the end of the list marked by a continuation variable—the continuation variable α *is* the delimiter of the functional context.

The symmetrical structural substitution rules in $\lambda\mu$-calculi can be compared to the symmetrical μ and $\tilde{\mu}$ operators. Recall that in $\lambda\mu$-calculi, the passivate term $[\alpha]M$ can be used to mark arbitrary subterms, and the activate term $\mu\alpha.N$ gives a 'handle' to those subterms at a higher level.

A function application in the $\lambda\mu$-calculus with the activate term in the function position (indicated by the rule $(\mu\text{-}structural_R)$) represents the action of supplying an argument to the marked function. This can be achieved in $\overline{\lambda}\mu\tilde{\mu}$ using the μ operator. In the simplest case, we have some term t marked with the name α. An

argument E can be supplied to t via α using the (μ)-rule.

$$\langle \mu\alpha.\langle t \parallel \alpha\rangle \parallel E\rangle \rightarrow^* \langle t \parallel E\rangle$$

Symmetrically, a function application with the activate term in the argument position (indicated by the rule $(\mu\text{-}structural_L)$) represents the action of applying a function to the marked subterm argument. This can be achieved in $\overline{\lambda}\mu\tilde{\mu}$ using the $\tilde{\mu}$ operator. In the simplest case, we have a placeholder x interacting with some evaluation context e. A function F can be applied to e via x using the $(\tilde{\mu})$-rule.

$$\langle F \parallel \tilde{\mu}x.\langle x \parallel e\rangle\rangle \rightarrow^* \langle F \parallel e\rangle$$

In spite of these computational insights, there are a few points about the $\overline{\lambda}\mu\tilde{\mu}$ calculus that makes the Curry-Howard correspondence awkward. Two axiom rules are needed to type the continuation variables and term variables, not every cut is a redex, and the structural rules involving the stoup are of a curious form. The authors remark that a correspondence with Gentzen's LK could be obtained by studying a subsyntax of $\overline{\lambda}\mu\tilde{\mu}$. This subsyntax is studied by Lengrand [61, 62] in some detail, which we review in the next section.

2.3.6 Lengrand's $\lambda\xi$-calculus

Lengrand [61] is interested in the Curry-Howard isomorphism for Classical Logic, and in particular studying reduction and the connection between explicit substitutions and cut elimination in the Sequent Calculus.

It is well known that the (natural deduction) cut-rule can be used to type explicit substitutions. The symmetry of the cut-rule in Gentzen's Sequent Calculus, LK, inspires Lengrand to study a symmetrical notion of explicit substitution. He begins with the subsyntax of $\overline{\lambda}\mu\tilde{\mu}$ suggested by Curien and Herbelin in [32], shown below, that corresponds to proofs in Gentzen's LK[13].

$$M ::= \langle x \parallel \alpha\rangle \mid \langle y \parallel \mu\alpha.M_1 \cdot \tilde{\mu}x.M_2\rangle \mid \langle \lambda x.\mu\alpha.M \parallel \beta\rangle \mid \langle \mu\alpha.M_1 \parallel \tilde{\mu}x.M_2\rangle$$

Despite each of these constructs being built using commands, they are not 'computational' commands in the sense of $\overline{\lambda}\mu\tilde{\mu}$ reductions. Only the last construct

[13] In fact, Lengrand reformulates this and defines his own syntax, but this is again later reformulated when $\lambda\xi$ is adopted as a basis for the $\mathcal{X}$-calculus. The interested reader can refer to [61, pp. 24] for Lengrand's own syntax.

is typed with the cut, the rest are typed (from left-to-right) with the axiom, left implication introduction and right implication introduction rules.

Lengrand questions the reading of the symmetrical construct $\langle \mu\alpha.M_1 \parallel \tilde{\mu}x.M_2 \rangle$: "the input x of M_2 is replaced by the output α of M_1 (or is it the output α of M_1 that is replaced by the input x of M_2?)". He is able to give an answer to his question by defining two reduction systems called (respectively) call-by-name and call-by-value systems. The reduction systems themselves are step-wise cut-elimination procedures that corresponds to a fine grained explicit substitution mechanism.

In addition to the symmetries exhibited by $\overline{\lambda}\mu\tilde{\mu}$, Lengrand introduces an additional notion to his term calculus, in accordance with Gentzen's original cut-elimination procedure. He observes that there are in fact *two* possible reductions for the interaction between a function and evaluation context (corresponding to the two permutations of the cut-elimination rule for implication). These rules correspond to the two versions of the (*exp-imp*) rule in the $\mathcal{X}$-calculus (see Definition 3.1.4). The important point to note is that one version belongs to a call-by-name system while the other belongs to a call-by-value.

Summers [83] notices that the $\overline{\lambda}\mu\tilde{\mu}$ reduction mechanism incorporates only one of the variants of the cut-elimination rules for implication. According to Lengrand's definition of call-by-name and call-by-value, $\overline{\lambda}\mu\tilde{\mu}$ is therefore biased towards a call-by-value reduction. Summers proposes the alternative and symmetric formulation of the $\overline{\lambda}\mu\tilde{\mu}$ rule $(\rightarrow)$ following a remark of Herbelin [32, 47].

Definition 2.3.24 (Alternative Reduction rules for $\overline{\lambda}\mu\tilde{\mu}$ [83, 47]) *The reduction rules from Definition 2.3.19 can alternatively be augmented with the following rules.*

$$(\rightarrow_{CBV}) : \ \langle \lambda x.\mu\alpha.c \parallel v \cdot e \rangle \rightarrow \langle v \parallel \tilde{\mu}x.\langle \mu\alpha.c \parallel e \rangle \rangle$$
$$(\rightarrow_{CBN}) : \ \langle \lambda x.\mu\alpha.c \parallel v \cdot e \rangle \rightarrow \langle \mu\alpha.\langle v \parallel \tilde{\mu}x.c \rangle \parallel e \rangle$$

Notice that $(\rightarrow_{CBV})$ gives preference to reducing the argument v of the function ahead of evaluating the body of the function. In other words, it prefers to provide any inputs to the function. In contrast, the rule $(\rightarrow_{CBN})$ prefers to provide the output e to the function, before its evaluation.

In [62], Lengrand and Lescanne present an alternative formulation of $\lambda\xi$, using the strongly normalising cut-elimination procedure of Urban [86]. Urban's cut-elimination procedure is the first that jointly satisfies three important criteria: (i) strong normalisation, (ii) preservation of 'essential normal forms' and (iii) cut-over-cut permutation (necessary to simulate β-reduction).

The $\lambda\xi$ calculus restores the 'cut=redex' paradigm, eliminates the need for the stoup and additionally requires only one axiom rule to type variables (which are of the shape $\langle x \parallel \alpha \rangle$). Van Bakel and Lescanne [9] saw this work as a promising line to follow to obtaining a full Curry-Howard correspondence for Classical Logic. Together, they designed a more symmetrical syntax which used the hat notation of Whitehead and Russell [76] to represent binding (i.e. $\hat{}$). Of course, since this calculus no longer featured the λ binder, it was appropriately renamed; the $\mathcal{X}$-calculus is studied in Chapter 3.

2.3.7 Reduction Subsystems and Strategies

The pure λ-calculus gives an unrestricted definition of β-reduction (see Definition 2.3.4). This presentation of the calculus is inherently *non-deterministic*: given a program not in normal form, there is often a choice of which redex should be evaluated next. For terminating executions, the *confluence* of the calculus ensures the chosen order of these redexes is irrelevant to the outcome of the program.

Plotkin, in [70], defined two *subreduction systems* for the λ-calculus, namely the call-by-name (CBN) and call-by-value (CBV) subsystems. A subreduction system restricts the reduction system of a calculus and redefines the form of reducible expressions. In the λ-calculus, Plotkin's subsystems, defined below, impose restrictions on the applicability of the β-reduction rule. The restriction requires a subset of the λ-terms to be readily identifiable; this is achieved by formulating a modified grammar, given below.

Definition 2.3.25 (λ-calculus Modified Grammar) *An equivalent formulation of the language given in Definition 2.3.1 is given below, making the distinction of a* value *(denoted V) in the λ-calculus.*

$$\begin{aligned} M, N &:= (MN) \mid V \\ V &:= x \mid (\lambda x.M) \end{aligned}$$

Definition 2.3.26 (Call-by-Name to Weak Head Normal Form) *The call-by-name subsystem $\xrightarrow{cbn}$, defined by the following operational rules, imposes the following restrictions on the applicability of the β-reduction rule for the λ-calculus.*

$$(\lambda x.M)N \xrightarrow{cbn} M\{N/x\} \tag{2.3}$$

$$\frac{M \xrightarrow{cbn} N}{(MP) \xrightarrow{cbn} (NP)} \tag{2.4}$$

Definition 2.3.27 (Call-by-Value to Weak Normal Form) *The call-by-value* $\xrightarrow{cbv}$ *subsystem is defined by the following operational rules.*

$$(\lambda x.M)V \xrightarrow{cbv} M\{V/x\} \tag{2.5}$$

$$\frac{M \xrightarrow{cbv} N}{(MP) \xrightarrow{cbv} (NP)} \tag{2.6}$$

$$\frac{M \xrightarrow{cbv} N}{(VM) \xrightarrow{cbv} (VN)} \tag{2.7}$$

The call-by-name system 'lazily' applies arguments to abstractions whether or not those arguments have been normalised first; this has the effect that a substituted argument may be evaluated more than once if it is copied during the substitution. In contrast, the 'eager' call-by-value system requires that all arguments are the shape of values V before being applied to an abstraction, i.e., a value may only be substituted for a formal parameter. This means that in a call-by-name system, the term $\lambda x.((\lambda y.y)(xz))$ is reducible, whereas in a call-by-value system it would be considered to be in normal form.

Despite being defined operationally by Plotkin, there is some disagreement over the precise definition of the CBN and CBV subsystems. In [79], Sestoft makes the point that programs of real world programming languages (e.g. Haskell, Scheme and Standard ML) cannot have free variables and that reductions are not performed under lambda abstractions for reasons of efficiency. This leads to terms which although being the result of a computation are in a 'weaker' kind of normal form when compared to the unrestricted calculus. He argues that in the context of the λ-calculus, such reductions *must* be performed, for otherwise the 'addition' of Church Numerals would not yield the correct results ("which would disappoint students").

A commonly adopted approach is to relax the operational rules and allow reduction to continue 'past' the weaker normal form—thereby preserving the essential features of the original subsystem. The *normal-order* and *applicative-order* subsystems are (respectively) the counterparts of the call-by-name and call-by-value subsystems that attempt to fully normalise terms; their essential feature is to allow reductions under abstractions.

In [78], Selinger gave a categorical semantics to the call-by-name and call-by-

value subsystems of the $\lambda\mu$-calculus, and in this setting he noticed they were 'dual' subsystems. This observation is harder to see from the point of view of the pure λ-calculus—a side effect of its unsymmetrical notions of named input and anonymous output.

Sequent calculi exhibit these symmetries more naturally, and equivalent notions of call-by-name and call-by-value have been defined for the corresponding term calculi. Herbelin describes two confluent strategies for $\overline{\lambda}\mu\tilde{\mu}$: "giving priority to (μ) leads to a call-by-value language and giving priority to $\tilde{\mu}$ leads to a call-by-name language: in the critical pair $\langle \mu\alpha.c \parallel \tilde{\mu}x.c \rangle$ it is a call-by-name evaluation discipline if the evaluation context binds its argument as it stands to x, it is a call-by-value evaluation discipline if the evaluation context expects first its argument to be evaluated before binding it to x, which means yielding its priority to the term" [48]. Similar definitions of confluent strategies exist in $\lambda\xi$, and therefore also in the $\mathcal{X}$-calculus.

Although the call-by-name and call-by-value systems we will study are confluent, the computations of terms are not necessarily deterministic. Since computers are inherently deterministic, real-world implementations of programming languages impose a *strategy* on reduction in order to make computations deterministic. This gives a form of consistency: the same program will always run in the same way under the strategy. For example, viewing a call-by-name λ-calculus as a tree of subterms, a left-most outermost redex picking traversal of the tree will always yield the same reduction path. In our implementation of the $\mathcal{X}$-calculus (Chapter 4) we will impose strategies on reduction subsystems to obtain a deterministic reduction system.

2.4 Rewriting Higher-Order Terms

One of the research goals of this thesis was to study the reduction mechanism of the $\mathcal{X}$-calculus and its extensions. Our first observation was that if we were to view such calculi as simple term rewriting systems, several unconventional features are immediately apparent, the most conspicuous of these being:

1. The presence of binders.
2. Term constructors that bind several variables simultaneously.
3. Side-conditions on rewrite rules.
4. Non-confluent reduction.

5. An unconventional notion of 'substitution' (i.e., not 'term-for-variable' substitution).

The presence of binders in term rewriting systems escalates their class from being first-order to *higher-order*. We take this definition from van Raamsdonk's thesis [73] that defines such higher-order systems "...as rewriting systems in which a binding mechanism for variables is present".

We will present, in some detail, various approaches different researchers have taken to define higher-order rewrite systems, focusing on operational aspects.

2.4.1 Higher-Order Terms

In a higher-order calculus, such as the λ-calculus, every variable is either *free* or *bound*. The *binding relation* that associates the function parameters with its free occurrences in the function body is usually specified implicitly. In the λ-calculus for example, the symbol for function abstraction, λ, is always constructed with a named parameter. Occurrences of variables within the function body are associated with the binder whose parameter *name* is the same as that of the variable. This implicity also extends to the *variable identity relation* that equates two free variables: two 'equal' occurrences of a free variable are represented with the same name. Using the syntax in this way is intuitive and very readable, and so it is the preferred method for representing binding structure in terms.

However, implementors of languages with binding constructors must take care when working with free and bound variables. Implementations will require an explicit binding and variable identity relation to make the implicit notions explicit. As we will see, a variable-scoping convention and some form of administration to uphold the conventions in place will also be needed. Over-simplifying the situation, the following example highlights the need for these features.

$$\lambda y.(\lambda y.yy)$$

Without a scoping convention in mind, it is impossible to know which y is bound by which abstraction. Some seemingly 'obvious' solutions (which do not work) might be simply to pick fresh names for each abstraction and/or always read the terms so that each variable is bound to the closest binder. However, even under these conventions the example term above can be reached by repeatedly applying

the β-rule to the term $(\lambda w.ww)(\lambda xy.xy)$.

$$\begin{aligned}(\lambda w.ww)(\lambda xy.xy) &\rightarrow_\beta ww\{(\lambda xy.xy)/w\}\\ &= \quad (\lambda xy.xy)(\lambda xy.xy)\\ &\rightarrow_\beta \lambda y.xy\{(\lambda xy.xy)/x\}\\ &= \quad \lambda y.(\lambda xy.xy)y\\ &\rightarrow_\beta \lambda y.((\lambda y.xy)\{y/x\})\\ &= \quad \lambda y.(\lambda y.yy)\end{aligned}$$

Although the starting term of the reduction sequence was unambiguous, the final result was incorrect because no administration is in place to uphold the scoping convention during reduction. The parameter names of the binders λx and λy were duplicated when a copy of the term $\lambda xy.xy$ was made, creating the possibility of a later conflict.

The solution to the problem in the λ-calculus lies in specifying exactly how one should evaluate the meta-substitution operation $M\{N/x\}$. We remark that this is often not seen as a problem for mathematicians and many theorists, since the substitution is usually *defined* to be 'capture avoiding' and is performed instanteously; this is a good thing, since the purpose of the λ-calculus is to model functions and so should not be complicated with such mechanical aspects. However, for anyone wishing to implement these kinds of languages or study the reduction mechanism in finer detail, the exact substitution operation must be defined precisely.

A number of methods have been devised and due of the nature of the operation, they are largely mechanical. The proposals that have been made have striven towards efficiency, which becomes more important as the expression being reduced becomes more complicated. We will review some of the existing mechanisms that have been specified to evaluate substitutions and focus specifically on solutions that ensure the substitutions are 'capture avoiding'.

2.4.2 Explicit Substitutions

Perhaps the most intuitive mechanism is to consider the substitution as a separate syntactic structure, and then define step-wise how the substitution descends through the term tree. The $\lambda\mathbf{x}$-calculus of Bloo and Rose [18] does exactly this.

Definition 2.4.1 (λx [18]) *The set $\lambda\boldsymbol{x}$ is defined as follows:*

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

A term of the form $M\langle x = N\rangle$ is called a closure. *The definition of free variables is extended to include,*

$$fv(M\langle x = N\rangle) = fv(M)\backslash\{x\} \cup fv(N)$$

Definition 2.4.2 (Reduction in $\lambda\mathbf{x}_{gc}$ [18]) *The following reduction rules on $\lambda\boldsymbol{x}_{gc}$ terms are identified.*

$$\begin{array}{lllll}
(\beta'): & (\lambda x.M)P & \rightarrow M\langle x = P\rangle & & \\
\mathsf{(gc)}: & M\langle x = P\rangle & \rightarrow M & \leftarrow & x \notin fv(M) \\
\mathsf{(VarI)}: & x\langle x = P\rangle & \rightarrow P & & \\
\mathsf{(VarK)}: & y\langle x = P\rangle & \rightarrow y & \leftarrow & y \neq x \\
\mathsf{(App)}: & (MN)\langle x = P\rangle & \rightarrow M\langle x = P\rangle N\langle x = P\rangle & & \\
\mathsf{(Abs)}: & (\lambda y.M)\langle x = P\rangle & \rightarrow \lambda y.(M\langle x = P\rangle) & &
\end{array}$$

The rule (β') transforms the function application to the closure construct, which searchers (in a step-wise manner) for free occurrences of the formal parameter. The (VarI) rule replaces any free occurrences with the function argument.

With regards to preserving the *binding relation* (discussed in Section 2.4), the fine grained rules above make it easier to pinpoint where problematic reduction paths may arise.

We observe that the only potential conflict is in the rule (Abs), where the subterm P acquires y as a new binder—causing any free variables named y occurring in P to be *captured*. In addition, a variable *clash* will occur if the parameter names of the binders 'λx' and 'λy' are the same—causing the scopes of the two binders to be exchanged. A replacement rule (Abs') can be formulated which avoids these problems.

$$(\mathsf{Abs}'): \qquad (\lambda y.M)\langle x = P\rangle \rightarrow \lambda z.(M\langle y = z\rangle\langle x = P\rangle)\ z \textit{ fresh}$$

An α-conversion is performed on the abstraction whenever a closure is to be brought into its scope. Since a *fresh* binder name is used, there is no possibility that the variables of P will share the same name, and, there can be no name clash between the abstraction and the closure.

Although correct, this is an expensive operation to perform and will most likely be unnecessary in the majority of cases. The situation can be improved slightly if one allows the use of side-conditions on rules, so that α-conversions are done only when there will be a variable capture or clash, i.e.,

$$\begin{array}{lll} (\mathsf{Abs}''): & (\lambda y.M)\langle x=P\rangle \rightarrow \lambda z.(M\langle y=z\rangle\langle x=P\rangle) & \leftarrow\ y \in fv(P) \vee y{=}x \\ (\mathsf{Abs}'''): & (\lambda y.M)\langle x=P\rangle \rightarrow \lambda y.(M\langle x=P\rangle) & \leftarrow\ y \notin fv(P) \wedge y{\neq}x \end{array}$$

However, we remark that the true cost of the reduction is now hidden, and dependant on the implementation of the free-variable side conditions. Many researchers choose to simply ignore the cost of side-conditions in benchmarks, but as we will see later the cost of computing this set increases as the size of the subterm increases.

2.4.3 De Bruijn Indices

A notable problem of using names to represent the binding and variable identity relations is in comparing terms for equality. We can intuitively recognise that the two lambda-terms $\lambda x.x$ and $\lambda y.y$ are equal (i.e., an argument applied to either function will yield the same result) because we realise the names of the binders are irrelevant; the important feature is the *term structure* that has been encoded using the names. A machine testing the equality of these two terms would not consider them equal on lexical syntax alone. Some mapping between the implied structure of the two terms would need to be involved in the equality check.

De Bruijn's introduced a 'nameless' calculus, $C\lambda\xi\phi$, in [23]; nowadays it is referred to simply as 'de Bruijn Indices'. The calculus uses natural numbers rather than names to encode term structure and has the advantage that terms are unique within a particular α-conversion class. For example, both $\lambda x.x$ and $\lambda y.y$ would be represented by the same term (i.e., $\lambda 1$). The basic intuition behind the syntax is that the natural number is an explicit reference to its binder; the magnitude of the number is a count of the lambda binders traversed *upwards* through the term syntax tree to the binder. The language of $C\lambda\xi\phi$ is presented below.

Definition 2.4.3 (de Bruijn Terms, $C\lambda\xi\phi$ [23]) *We use $\mathbb{N}$ to represent the set of positive natural numbers and define the set $C\lambda\xi\phi$ as,*

$$M, N ::= \mathbb{N} \mid \lambda M \mid MN$$

We will use $a, b, \ldots$ to range over the set $C\lambda\xi\phi$, and $i, j, m, n, \ldots$ to range over $\mathbb{N}$.

The substitution mechanism, that corresponds to β-reduction in the λ-calculus, is specified by the following rule.

$$(\beta''): (\lambda a)b \rightarrow a\{1 \leftarrow b\}$$

The right-hand side of the rule utilises a meta-operation that replaces any indices in a referring to the λ with a copy of the term structure specified by b. Each such reference in a is an index whose value corresponds to the number of binders on the path from the λ to itself—the point being that indices, with different numbers may refer to the same λ. For example, in the term $\lambda 1(\lambda 2)$ (corresponding to $\lambda x.x(\lambda y.x)$), both indices although different in value, refer to the outermost λ.

The reduction relation is *compatible*. Therefore, to compute the meta-operation $\{i \leftarrow b\}$ over the term a, the magnitude of any indices in the function's body a that point to the enclosing context will need to be decremented, since a binder was destroyed in the contraction of the redex. There may also be references within b to the surrounding context, and since copies of b substituted into a may acquire any number of new binders, those references must also be updated to preserve the structure of the original argument—this operation is known as *lifting*. The evaluation of $a\{i \leftarrow b\}$ is therefore split into three parts: (i) locate the references to the λ-abstraction in the function body, (ii) update the function body to take account of the contracted redex (iii) substitute the function argument into the relevant locations, lifting where relevant. Parts (i) and (ii) are computed as follows.

$$\begin{aligned}
(a_1 a_2)\{i \leftarrow b\} &= (a_1\{i \leftarrow b\})(a_2\{i \leftarrow b\}) \\
(\lambda a)\{i \leftarrow b\} &= \lambda(a\{(i{+}1) \leftarrow b\}) \\
n\{i \leftarrow b\} &= \begin{cases} n{-}1 & \text{if } n{>}i \\ \uparrow_0^i(b) & \text{if } n{=}i \\ n & \text{if } n{<}i \end{cases}
\end{aligned}$$

The rule (β'') works as follows. As the construct $\{1 \leftarrow b\}$ traverses the term structure of a, a counter (starting from 1) records the number of binders that have been traversed. When an index is encountered, its value will be less than the counter if it refers to a binder within the function body, and greater than the counter if it points to a binder to the surrounding context. Indices equal to the counter are references to the λ binder of the redex (i.e., the binder being sought). The lifting operation $\uparrow_j^i(b)$ ensures any indices in copy of b that refer to a binder in the surrounding context are updated to account for any new binders acquired. The

lifting operation is defined below.

$$\begin{aligned}
\uparrow^i_j(a_1\ a_2) &= \uparrow^i_j(a_1)\ \uparrow^i_j(a_2) \\
\uparrow^i_j(\lambda a) &= \lambda(\uparrow^i_{j+1}(a)) \\
\uparrow^i_j(n) &= \begin{cases} i & i<j \\ i+n-1 & \textit{otherwise} \end{cases}
\end{aligned}$$

Example 2.4.4 (Example Reduction using De Bruijn Indices) *The mechanism for simulating substitutions performed by the evaluation of $\{i \leftarrow b\}$ is carried out instantaneously. The following example reduction of the term $\lambda y.\underline{(\lambda xy.xy)y}$, which is represented as $\lambda(\lambda\lambda 21)1$ in de Bruijn indices notation, is shown in full detail below.*

$$\begin{aligned}
\lambda(\lambda\lambda 21)1 &\rightarrow_{\beta''} \lambda((\lambda 21)\{1 \leftarrow 1\}) \\
&= \lambda(\lambda((21)\{2 \leftarrow 1\})) \\
&= \lambda(\lambda(2\{2 \leftarrow 1\})(1\{2 \leftarrow 1\})) \\
&= \lambda(\lambda(2\{2 \leftarrow 1\})(1)) \\
&= \lambda(\lambda(\uparrow^2_0(1))(1)) \\
&= \lambda(\lambda(21))
\end{aligned}$$

In λ-calculus notation, the result of the computation is the λ-term $\lambda y.(\lambda z.yz)$.

The example shown above is easy to understand, but as terms get larger the complexity quickly increases. The use of de Bruijn indices has its advantages and disadvantages, which we summarise below.

Advantages of Using de Bruijn Indices

Most notably, all α-equivalent terms are represented by the same syntax, making a static equality check of two terms trivial. Although during the reduction of a term, the binding relation (encoded by the indices) will still to be administered, the use of de Bruijn indices effectively reduces this task to arithmetic operations which are performed very efficiently by machines. In comparison, generating fresh variable names as would be required in a named calculus is expensive.

De Bruijn indices are particularly well suited to the substitution of variables by closed expressions (e.g., used in inlining of functions). In these cases, the lifting operation is not needed at all, since there will be no references to the enclosing context.

Each λ effectively declares a new 'scope'. Since each index is essentially a count of the enclosing scopes, an implementation can be achieved very intuitively using stacks. An interpreter can use an 'environment stack' besides the usual execution stack to associate the value with each variable in the current scope. A variable's value is found by simply popping the stack the number of times indicated by the index. Garbage collection of environments is also straightforward: when a new scope is entered the value for the variable is pushed onto the stack, and when the scope is exited, the out-of-scope variables are popped.

Disadvantages of Using de Bruijn Indices

Implementations of the λ-calculus using de Bruijn indices are notoriously difficult to debug due to unhelpful compiler error messages. While a language of numbers is easily parsable by machines, it is impractical for humans to comprehend. Notice at the end of the Example 2.4.4, we *chose* to represent the outer variable y with the same name before and after the contraction of the redex; a computer would not easily make such a choice. Various researches however have devised more general systems which allow a relation between names and indices to be maintained during a reduction (see for example [66, 81]).

It is often stated that by employing de Bruijn indices as a method of implementation, α-conversions are not required and variable capture/clash is automatically avoided. What is not often stated is that these properties do not come for free. In a named calculus, α-conversions are performed administratively to uphold the binding relation. Using de Bruijn indices, this administrative work is still done, except it takes the form of the arithmetic operations previously discussed—these operations essentially mimic the α-conversion steps. When the function argument contains many indices referring to binders in the enclosing context, it is possible that many more administrative steps may be required using de Bruijn's notation. Consider an example in a named calculus where the free variables in the argument of the redex have names that will not cause any variable capture or clash. Using de Bruijn indices the lifting will need to be performed regardless.

More importantly, parallel β-reduction is also problematic for non-closed terms, since one needs to take into account indices within the redex that refers to the enclosing context.

2.4.4 Wadsworth's λ-graphs

Wadsworth was the first to use graphs as a means of implementation for the λ-calculus [92]. Graphs structures allow for the *sharing* of common sub-expressions which naturally leads to a space (memory) efficient implementation, and time efficiency where parallel reductions are permitted.

Instances of Wadsworth's λ-graphs are built from a universe, U, of objects called nodes, of which there are three types: those for application (γ), abstraction (λ) and identifiers (τ). The type is used to associate a finite number of predicates, selector functions and data components with each node. These associations are summarized in the table below.

Object	Type	Predicate	Selectors	Data Components	
			$U \rightarrow U$	Name	Nature
Application	γ	Isγ	*rator*, *rand*	–	–
Abstraction	λ	Isλ	*body*	*bv*	*identifier*
Identifier	τ	Isτ	–	*var*	*identifier*

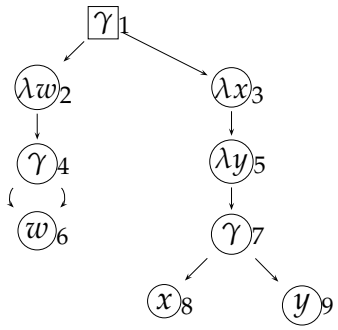
Definition 2.4.5 (λ-graph [92]) *A well formed λ-graph is specified by the quintuple $\langle N, s_1, s_2, s_3, z\rangle$ where,*

- *N is a finite set of nodes in U; we will use numbers to represent elements of U and for illustrative purposes, we will sometimes annotate elements of U with node types $\{\gamma, \lambda, \tau\}$. Such annotations will always explicitly specify the data-components of each node (e.g., the annotated node objects for application, abstraction and identifiers may be written as $1{:}\gamma$, $2{:}\lambda x$ and $3{:}\tau x$ respectively).*
- *s_1 and s_2 are functions from N to N defined as the (ope)rator and (ope)rand selectors for a node if the predicate Isγ holds.*
- *$s_3 : N \rightarrow N$ is defined as the 'body' selector for all nodes in N if the predicate Isλ holds.*
- *$z \in N$ is the unique root node of the graph.*

Example 2.4.6 (λ-graph Structure) *The λ-graph, $G{=}\langle N, s_1, s_2, s_3, z\rangle$ for the λ-term $(\lambda w.ww)(\lambda xy.xy)$ is:*

$$
\begin{aligned}
N &= \{1{:}\gamma, 2{:}\lambda w, 3{:}\lambda x, 4{:}\gamma, 5{:}\lambda y, 6{:}\tau w, 7{:}\gamma, 8{:}\tau x, 9{:}\tau y\} \\
s_1 &= \{(1,2),(4,6),(7,8)\} \\
s_2 &= \{(1,3),(4,6),(7,9)\} \\
s_3 &= \{(2,4),(3,5),(5,7)\} \\
z &= 1
\end{aligned}
$$

In diagrammatic form, directed edges are used to represent the selector functions and circles are used to represent node objects; a square node is used to identify the unique root node z.



An interesting feature of Wadsworth's graphs is that the binding relation is *not* reflected in the structure of the graph. Notice that in the definition of λ-graphs (Definition 2.4.5), there is no association between the identifier nodes and the abstraction nodes. To resolve this, Wadsworth defines a series of functions to compute the binding relation based on the graph's structure and on the data-components of the graph nodes.

Definition 2.4.7 (Path [92]) *A* path *is a list of $n \geq 0$ selectors. If $p=[s_1, s_2, \ldots, s_n]$ is a path of length >0, then the first selector s_1 and the list $[s_2, \ldots, s_n]$ are called the head and tail of p.*

If $p'=[s'_1, \ldots, s'_m]$ is a second path, then $p \bullet p'$ denotes the path $[s_1, s_2, \ldots, s_n, s'_1, \ldots, s'_m]$.

If p and q are two paths, then p is a stem *of q if there is a path p' such that $(p \bullet p')=q$.*

The application of a path p to a node object o is written $p(o)$ whose result is the composition of selectors applied to the argument node, i.e., the node object $s_m(\ldots(s_2(s_1(o))))$

Definition 2.4.8 (Binding Relation [92]) *Given a graph $\langle N, s_1, s_2, s_3, z\rangle$, we first define the following partitions over the finite set of graph nodes N:*

$$\begin{aligned} &\textit{Application Nodes,} && A = \{n \in N \mid Is\gamma(n)\} \\ &\textit{Abstraction Nodes,} && L = \{n \in N \mid Is\lambda(n)\} \\ &\textit{Identity Nodes,} && T = \{n \in N \mid Is\tau(n)\} \end{aligned}$$

Following Wadsworth's original definitions, some auxiliary functions need to be given before we can define the binding relation. In the following functions defined using set

comprehensions, we use the variables p, q to denote paths, n to denote nodes and G to denote λ-graphs.

$$\begin{aligned}
\mathit{NodesOn}(p,n,G) &= \{n' \in N \mid q(n){=}n' \text{ for some stem } q \text{ of } p\} \\
\mathit{BvsOn}(p,n,G) &= \{\mathit{bv}(l) \mid l \in (L \cap \mathit{NodesOn}(p,n,G))\} \\
\mathit{FreeOccs}(x,n,G) &= \{t \in T \mid \mathit{var}(t) \notin \mathit{BvsOn}(p,n,G) \text{ for some } p \text{ with } p(n){=}t\} \\
\mathit{FreeNodes}(n,G) &= \{t \in \mathit{FreeOccs}(\mathit{var}(t),n,G)\} \\
\mathit{FreeVars}(n,G) &= \{\mathit{var}(t) \mid \text{some } t \in \mathit{FreeNodes}(n,G)\}
\end{aligned}$$

Now we can define the binding relation over the λ-graph $G{=}\langle A \cup L \cup T, s_1, s_2, s_3, z\rangle$.

$$\begin{aligned}
\mathit{BinderRel(G)} = \{(l,t) \mid l \in L \wedge t \in \mathit{FreeOccs}(\mathit{bv}(l),\mathit{body}(l),G)\} \\
\cup \{(\star,t) \mid t \in \mathit{FreeOccs}(z,G)\}
\end{aligned}$$

The binding relation is a set of tuples where the elements of each tuple refer to a λ node and an identity node respectively. Each globally free variable node t in the graph is assigned a unique binder written $\star$.

We recall that the binding relation must always be maintained by some administrative work during reduction. Wadsworth's approach is to restrict reduction to operate on only graphs which are of a special shape—he calls these graphs *R-admissible* graphs.

Definition 2.4.9 (Admissible and R-admissible Graphs [92]) *A λ-graph is* admissible *if it is acyclic and each terminal node has a unique binder. An R-admissible graph is an admissible graph that has a single pointer to the rator node (the abstraction) of a β-redex.*

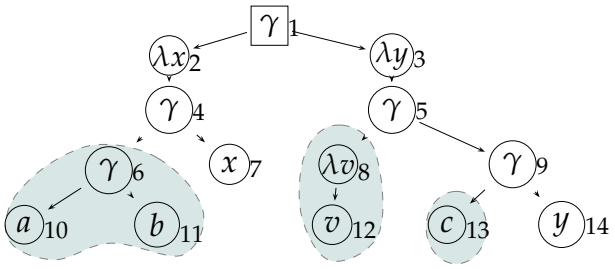
Admissibility defines a strong ownership property between the abstraction nodes and the identifier nodes they bind over, while the R-admissibility property restricts the sharing that can take place in the graph. If the rator node of a redex is shared, a copy of the subgraph at the rator node must be made before reduction can proceed (thus, turning an admissible graph into an R-admissible one). The cost of computing this operation can be lowered by identifying regions of the subgraph which do not need to be copied. To that end, Wadsworth defines *directly abstractable* subgraphs.

Definition 2.4.10 (Directly Abstractable) *Given a graph $G{=}\langle N, s_1, s_2, s_3, z\rangle$, a subgraph of G is a graph $G'{=}\langle N, s_1, s_2, s_3, z'\rangle$ where $p(z){=}z'$ for some path p. The subgraph*

G' is said to be directly abstractable *from a graph G iff $BvsOn(p, z, G) \cap FreeVars(z', G)$ is empty for all paths p, with $p(z)=z'$. Any subgraph of z' is said to be* abstractable *from G'.*

The intuition behind this operation is to identify the regions of the graph that do not depend on the function's argument. Such regions can essentially be 'taken out' of the function body (by use of an outer λ-type node for example), thus sharing of these regions can remain intact.

Example 2.4.11 *The λ-graph for $(\lambda x.(ab)x)(\lambda y.(\lambda v.v)(cy))$ is shown below. The subgraph at node 6 is directly abstractable from node 2, while subgraphs at nodes 8 and 13 are directly abstractable from node 3.*



To obtain an R-admissible graph from a 'non'-R-admissible graph, a *Copy* procedure is applied to the rator node of a redex. This copies out regions of the graph that are *not* abstractable from the rator of the redex.

Definition 2.4.12 (Copy Algorithm [92]) *Let $G=\langle N, s_1, s_2, s_3, z\rangle$ be an* admissible *graph with a redex-node r, involving the abstraction $l=s_1(r)$. The algorithm given below describes the steps that copy the parts of the redex that are not directly abstractable, building an R-admissible copy of r.*

1. *Construct the set C which identifies the graph nodes that need to be copied.*

$$\begin{aligned} S &= \text{the set of sub-nodes of } l. \\ C &= \{l\} \cup \{n \in S \mid n \text{ not abstractable from } l\} \end{aligned}$$

2. *Build a set C' of* copies *(identical type and data components but indexed differently) of nodes in C, and define a function h that maps each node of C to its copy in C'; the inverse of h will be written h^{-1}.*

3. *Define functions* $f : N \rightarrow ((N-C) \cup C')$ *and* $g : (N \cup C') \rightarrow N$ *where,*

$$f(n) = \begin{cases} h(n) & \text{if } n \in C \\ n & \text{otherwise} \end{cases}$$

$$g(n) = \begin{cases} h^{-1}(n) & \text{if } n \in C' \\ n & \text{otherwise} \end{cases}$$

4. *Define* $G'=Copy(r, G)=\langle N \cup C', s'_1, s'_2, s'_3, z\rangle$ *as the R-admissible copy of G where,*

$$\begin{aligned} \text{for all } i \in \{2,3\},\ s'_i(r) &= s_i(r) \\ \text{for all } i \in \{1,2,3\},\ s'_i(n) &= \begin{cases} f(s_i(g(n))) & \text{if } n \in C' \\ s_i(n) & \text{if } n \in N \wedge n \neq r \end{cases} \\ s'_1(r) &= f(l) \end{aligned}$$

With these definitions in place, the capture-free contraction operation on λ-graphs can be defined.

Definition 2.4.13 (Contraction of Redex Nodes [92]) *For a redex node r of an admissible graph* $G=\langle N, s_1, s_2, s_3, z\rangle$*, the contraction of a redex is computed by the following steps.*

1. *Form an R-admissible graph,* $G'=Copy(r, G)=\langle N', s'_1, s'_2, s'_3, z\rangle$.
2. *Ensure the bound variables of all* λ*-nodes are pairwise distinct and all different from the free variables of the graph; this can be achieved by the following steps.*
   (a) *Let* $v_1, v_2, \ldots, v_n$ *be the* λ*-nodes of r, and let* $v'_1, v'_2, \ldots, v'_n$ *be distinct variables not occurring in the graph.*
   (b) *For* $j = 1, 2, \ldots, n$*, change* $bv(v_j)$ *to* v'_j.
   (c) *For each terminal node with* $b(x) \neq \star$*, change its var-component to the bv of its binder.*
3. *Identify the key components of the redex.*

the function,	$f=s'_1(r)$
the formal parameter,	$v=bv(f)$
the function body,	$m=s'_3(f)$
the function argument,	$a=s'_2(r)$

4. *Construct the following two sets. (The set V corresponds to those identity nodes*

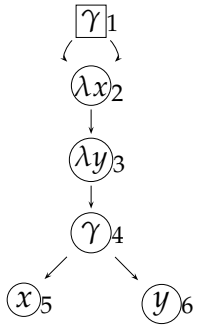
that are bound to the abstraction).

$$F = \textit{the set of sub-nodes of } f.$$
$$V = \{t \in F \mid Is\tau(t) \wedge var(t){=}v\}$$

5. *Adjust all pointers to r so they point to m.*
6. *Adjust all pointers to nodes in V so they point to a.*
7. *If r was the root of G, and Isτ(m), mark a as the new root; otherwise mark m as the new root.*

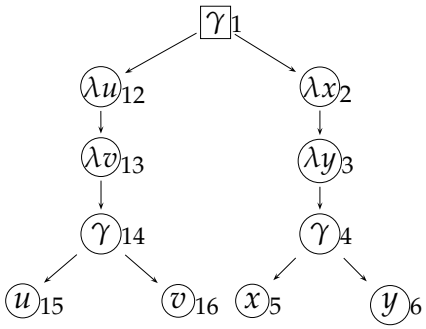
We will illustrate this process through an example reduction.

Example 2.4.14 (Reduction using λ-graphs) *The λ-term $(\lambda xy.xy)(\lambda xy.xy)$ is represented by the following λ-graph.*



$$G = \begin{cases} N = \{1,2,3,4,5,6\} \\ s_1 = \{(1,2),(4,5)\} \\ s_2 = \{(1,2),(4,6)\} \\ s_3 = \{(2,3),(3,4)\} \\ z = 1 \end{cases}$$

In the following, we will consider the normalisation of the graph G above according the reduction algorithm outlined in Definition 2.4.13. Observe that the redex r=1 is not an R-admissible graph. We therefore build the R-admissible graph G′=Copy(r, G) by copying out the nodes which are not *abstractable from r, and also make the bound variables pairwise distinct (i.e., ensure there are no two abstraction nodes in the graph whose bv data components are equal).*



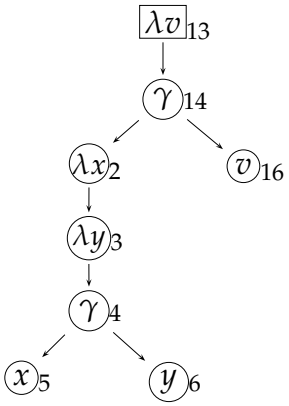
$$G' = \begin{cases} N' = \{1,2,3,4,5,6\} \cup \{12,13,14,15,16\} \\ s'_1 = \{(1,2),(4,5),(14,15)\} \\ s'_2 = \{(1,2),(4,6),(14,16)\} \\ s'_3 = \{(2,3),(3,4),(12,13),(13,14)\} \\ z' = 1 \end{cases}$$

The graph G′ is R-admissible, and so we may now identify the components of the redex

and compute the set of identity nodes V bound to the abstraction.

$$
\begin{aligned}
&f{=}12,\ v{=}u,\ m{=}13,\ a{=}2\\
&F = \{12, 13, 14, 15, 16\}\\
&V = \{15\}
\end{aligned}
$$

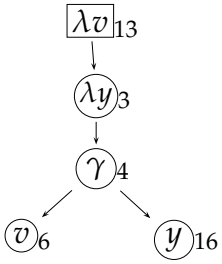
Follow the steps to contract the redex r=1, reassigning the root node of the graph appropriately, we obtain a λ-graph G'' corresponding to the λ-term $\lambda v.((\lambda xy.xy)v)$.



$$
G'' = \begin{cases}
N'' &= \{2, 3, 4, 5, 6, 13, 14, 16\}\\
s_1'' &= \{(14, 2), (4, 5)\}\\
s_2'' &= \{(14, 16), (4, 6)\}\\
s_3'' &= \{(13, 14), (2, 3), (3, 4)\}\\
z'' &= 13
\end{cases}
$$

This graph is also R-admissible and all bound variables are unique, so we may contract the final redex r=14, yielding the graph G''' corresponding to the λ-term $(\lambda vy.vy)$.

$$
\begin{aligned}
&f{=}2,\ v{=}x,\ m{=}3,\ a{=}16\\
&F = \{2, 3, 4, 5, 6\}\\
&V = \{5\}
\end{aligned}
$$



$$
G''' = \begin{cases}
N''' &= \{3, 4, 6, 13, 16\}\\
s_1''' &= \{(4, 6)\}\\
s_2''' &= \{(4, 16)\}\\
s_3''' &= \{(13, 3), (3, 4)\}\\
z''' &= 13
\end{cases}
$$

While Wadsworth was interested specifically in the computations concerned with the λ-calculus, other researchers in the field of term-rewriting noticed they could adapt the technique of λ-graphs to the be used in rewriting. Barendregt *et al.* [14] introduced a formalism that allowed one to implement first-order rewrite systems using a special kind of graph called 'term graphs'. We will study these kinds of graphs in the following section.

2.4.5 Term Graph Rewriting

Term graph rewriting was proposed as an extension of term rewriting to implement functional programming languages with some degree of efficiency [80, 14, 15]. The general idea is to consider transformations rules, known as *productions*, that act on graphs. A production is of the form $L \rightarrow R$, where L and R are the left- and right-hand sides of a map from the elements of a graph L to the elements of another graph R. A production defines precisely which graph elements must be preserved, created and destroyed during the application of a particular graph transformation. Since the terms are represented as graphs rather than trees, common subexpressions can be shared—leading to memory efficient structures. The reduction system is optimised since shared expressions need only be evaluated once.

We will recall here the basic notions and terminology.

Definition 2.4.15 (Term Graph Rewrite System) *A Term Graph Rewrite System or 'TGRS' is a pair $\langle \Sigma, R \rangle$, where Σ is a* signature *and R is a set of productions (or* reduction rules*) that transform term graph objects.*

Definition 2.4.16 (Signature , Σ) *The signature consists of:*

- *A countably infinite set of variables, $\mathcal{V}$.*
- *A non-empty set $\mathcal{F}$ of function symbols, where each function symbol is equipped with an* arity *specifying the number of arguments it can take.*

In this thesis, the term graph rewrite systems we will study operate on expressions represented as singly rooted ordered directed acyclic graphs. Such *term graphs* are defined below.

Definition 2.4.17 (Term Graph) *A rooted labelled graph over the signature Σ is a quadruple $\langle X, lab, succ, r \rangle$:*

- *$X \subset \mathbb{N}$ is a set of nodes.*
- *lab is a function of type $X \rightarrow \Sigma$, mapping each node in X to a symbol in Σ.*
- *succ is a function of type $X \rightarrow [X]$, specifying an ordered list of arguments for each node in X. The i^{th} argument of a node $n \in X$ with arity k is denoted $succ(n)_i$, where $0 \leq i < k$.*
- *r is the unique root of the graph.*

We do not require that every node is reachable from the root of the graph.

Below, we define subgraphs for term graphs which are analogous to subterms for terms.

Definition 2.4.18 (Path) *For a graph $\langle X, lab, succ, r\rangle$, an* annotated path *of length m (with $m \geq 0$) is a sequence $[n_0, i_1, n_1, \ldots, i_m, n_m]$ of nodes interleaved with integers, such that for all $0 \leq j < (m-1)$, we have $succ(n_j)_{i_j} = n_{j+1}$.*

A path *of length m from n_0 to n_m is a sequence of nodes $[n_0, \ldots, n_m]$, such that there exists an* annotated path *$n_0, i_1, \ldots, i_m, n_m$.*

Definition 2.4.19 (Subgraph) *Given an term graph $g = \langle X, lab, succ, n\rangle$, a subgraph $g|n$ of the graph g is the term graph $g = \langle X', lab', succ', n\rangle$ where*

$$X' = \{ n' \in X \mid \text{there is a path from } n \text{ to } n' \}$$

Every subgraph of a term graph is term graph.

With the above structures, we can define the term graph representation of a production called a *graph rewrite rule*.

Definition 2.4.20 (Open Graph) *An* open graph *is the triple $\langle X, lab, succ\rangle$ like a term graph, except no root is specified and the functions lab and succ are only required to be partial functions on X, with the same domain. A node for which lab and succ are undefined is referred to as an* open node *or* 0-ary metavariable*. When we write open graphs, we will specify a set of infinite symbols over which open nodes range. We say a graph is closed when it contains no open nodes.*

Definition 2.4.21 (Graph Rewrite Rule) *The graph representation of a production is a bi-rooted graph called a* graph rewrite rule*, and is denoted by the triple $\langle g, l, r\rangle$, where g is an open graph, and l and r are nodes of g called the* left root *and* right root *of the rule.*

The left- and right- hand sides of the rewrite rule are subgraphs, denoted $g|l$ and $g|r$.

Next we define a structure-preserving map known as a *homomorphism*. The map will be used to maintain a relation between the nodes of a graph rewrite rule and the nodes of a term graph. A homomorphism combined with a graph rewrite rule and a term graph define a *redex*.

Definition 2.4.22 (Homomorphism) *A homomorphism from a term graph* $\langle X, lab, succ, r\rangle$ *to a term graph* $\langle X', lab', succ', r'\rangle$ *is a map,* $\partial : X \rightarrow X'$*, where for all nodes in the set* $\{n \in X \mid$ *there is a path from* r *to* $n\}$*,*

$$\begin{aligned} lab'(x(n)) &= lab(n) \\ succ'(x(n)) &= x(succ(n)) \end{aligned}$$

We extend the definition to open labelled graphs by requiring the conditions specified above to hold only for nodes that are not open nodes.

Definition 2.4.23 (Redex) *A redex in a term graph* g_0 *is a pair* $\langle R, x\rangle$*, where* R *is a graph rewrite rule* $\langle g, l, r\rangle$ *and* x *is a homomorphism from* $(g|l)$ *to* g_0*. The homomorphism* x *is called an* occurrence *of* R *in* g_0*.*

With these definitions in mind, we now look at the steps performed during the graph transformation phases.

Definition 2.4.24 (Term Graph Substitution) *Given a term graph* $g{=}\langle X, lab, succ, r\rangle$*, and two nodes* $a \in X$ *and* $b \in X$*, the substitution operation* $g[a{:=}b]$ *is a term graph* $\langle X_s, lab_s, succ_s, r_s\rangle$ *defined as follows.*

$$\begin{aligned} X_s &= X \\ lab_s(m) &= lab(m) \\ succ_s(m)_i &= \begin{cases} b & \text{if } succ(m)_i = a \\ succ(m)_i & \text{otherwise} \end{cases} \\ r_s &= \begin{cases} b & \text{if } r = a \\ r & \text{otherwise} \end{cases} \end{aligned}$$

Definition 2.4.25 (Graph Rewrite Step) *Let* $\langle\langle g, l, r\rangle, x : (g|l) \rightarrow g_0\rangle$ *be a redex in the term graph* $g_0{=}\langle X_0, lab_0, succ_0, r_0\rangle$*. A* graph rewrite step *is an application of a graph rewrite rule to a term graph, where there is a unique homomorphism* x *from the nodes of the graph* $(g|l)$ *to the nodes of the graph* g_0*.*

A rewrite step transforms a term graph in three stages, called building, redirection and garbage collection; these stages construct the term graphs g_1*,* g_2 *and* g_3 *respectively.*

build : *An isomorphic copy of the nodes reachable from $(g|r)$ but not from $(g|l)$ are added to g_0. This set of nodes, $C{=}X_{(g|r)}{-}X_{(g|l)}$, is referred to as the 'copied nodes'. Simultaneously, the homomorphism x' is defined as an extension of x that additionally maps each node reachable from $(g|r)$ but not from $(g|l)$ to the copied nodes in X_{g_1}.*

We now define $g_1{=}\langle X_{g_1}, lab_{g_1}, succ_{g_1}, r_{g_1}\rangle$ where,

$$X_{g_1} = X_{g_0} \cup C$$

$$lab_{g_1}(m) = \begin{cases} lab_{g_0}(m) & \text{if } m \in X_{g_0} \\ lab_g(m) & \text{otherwise} \end{cases}$$

$$succ_{g_1}(m)_i = \begin{cases} succ_{g_0}(m)_i & \text{if } m{\in}X_{g_0} \\ succ_g(m)_i & \text{if } (m \in C) \wedge (succ_g(m)_i \in C) \\ x(succ_g(m)_i) & \text{otherwise} \end{cases}$$

$$r_{g_1} = r_{g_0}$$

redirect : *All edges in g_0 pointing to $x(l)$ are replaced by edges pointing to $x(r)$. More formally, $g_2 = g_1[x(l){:=}x(r)]$.*

garbage collect : *All nodes not accessible from the root of g_2 are removed. The term graph $g_3 = \langle X_{g_3}, lab_{g_3}, succ_{g_3}, r_{g_3}\rangle$ is defined as follows.*

$$\begin{aligned} X_{g_3} &= \{n \in X_{g_2} | \text{there is a path from } r_{g_2} \text{ to } n\} \\ lab_{g_3}(m) &= lab_{g_2}(m) \\ succ_{g_3}(m)_i &= succ_{g_2}(m)_i \\ r_{g_3} &= r_{g_2} \end{aligned}$$

2.4.6 Related Work

The term graph rewriting technique presented in the previous section expresses how to implement first-order term rewriting languages using directed acyclic graphs. As previously noted, first-order languages are unsuitable for implementing higher-order languages since, of course, they cannot express higher-order language features such as binding. In studying and implementing languages with such features, two approaches have been well studied:

1. The encoding of the binding relation in the data portions of nodes (e.g. de

Bruijn notation, Wadsworth's λ-graphs, conditional and copy term rewrite systems etc.)

2. Moving to a higher-order formalism that has an explicit representation of binding structures (e.g. Klop's combinatory reduction system [60], Khasidashvili's expression reduction systems [42], Kahl's second-order term graphs [52, 54] etc.).

Each of the works in the second approach extend the first-order systems by some means of variable-binding and term for variable substitution mechanisms. The substitution mechanisms are of the shape $N\{M/x\}$, where M and N are metavariables, and x is a bound variable (c.f. the variables used in λ-calculus notation). As we will see in Chapter 3, this does not correspond to the 'substitution-like' mechanism of computation present in the $\mathcal{X}$-calculus. As a result, in Chapter 4, we present a formalism that combines notions from both approaches. We will however review some of the features of Kahl's formalism, since it is a nice system to make comparisons with—in particular we will look at the *binding structure* encoded by Kahl's second-order graphs.

Kahl's Second-Order Term Graphs

Combinatory Reduction Systems (CRS) due to Klop [60] are a generalisation of first-order term rewrite systems with an explicit mechanism for binding metavariables. Kahl observes that the term *graph* rewrite systems in literature implement only conventional TRS', and so he formalised an implementation of CRS using term graphs with bound variables [52, 54]. The second-order term graph rewrite system he develops caters for cycles and provides a solution to the problem of variable capture.

A second-order term graph in Kahl's system is defined over an alphabet (c.f., a signature).

Definition 2.4.26 (Term Graph Alphabet [53]) *A second-order term graph is built from the alphabet* $(\mathcal{N}, arity, \mathcal{C}, \mathcal{B}, \mathcal{M})$*, with:*

$\mathcal{N}$	*the set of node labels, partitioned into disjoint sets* $\mathcal{C}, \mathcal{B}, \mathcal{M}$
$arity$	*the arity function*
$\mathcal{C}$	*the set of labels for 'constant constructors' (c.f. function symbols)*
$\mathcal{B}$	*the set of labels for bindable variables*
$\mathcal{M}$	*the set of labels for metavariables*

A key feature of Kahl's formalism, as opposed to Wadsworth's and de Bruijn's for example, is the distinction between the *structure* and *content* parts of terms. The variable binding and variable identity relations are made explicit in the definition of term graphs; an idea first attributed to Bourbaki [21] who investigated closed λ-terms[14]. Kahl extends this work additionally including a primitive variable identity relation that relates unbound variables in a second-order graph setting. Of course, by making the relation explicit in the structure of the graph, one dispenses with the need for variable *names* (since the purpose of the names was to imply these relationships).

Definition 2.4.27 (Term Graph [53]) *A term graph is the tuple* $(\mathcal{V}, label, succ, D, B, W)$*, with:*

$\mathcal{V} \subset \mathbb{N}$	*the finite node set*
$label : \mathcal{V} \to \mathcal{N}$	*the node labelling function*
$succ : \mathcal{V} \to [\mathcal{V}]$	*the ordered successor function*
$D : \mathcal{V} \leftrightarrow \mathcal{V}$	*the set of edges (for convenience, but implied from the successor function)*
$B : \mathcal{V} \nrightarrow \mathcal{V}$	*the binding function*
$W : \mathcal{V} \nrightarrow \mathcal{V}$	*the variable identity*

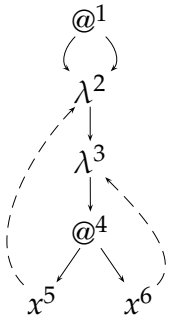
Roots in term graphs are considered with respect to D. The symbols $\leftrightarrow$*,* $\to$ *and* $\nrightarrow$ *to represent homsets, total functions and partial functions respectively.*

Example 2.4.28 (λ-calculus structures) *The graph alphabet for λ-calculus is given below.*

$$\begin{aligned} \mathcal{N}_{\geq} &= \{@, \lambda, x, V_0, V_1\} \\ arity_{\lambda} &= \{(@, 2), (\lambda, 1)\} \\ \mathcal{C}_{\geq} &= \{@, \lambda\} \\ \mathcal{B}_{\geq} &= \{x\} \\ \mathcal{M}_{\geq} &= \{V_0, V_1\} \end{aligned}$$

The λ-term $(\lambda x.\lambda y.xy)(\lambda x.\lambda y.xy)$ is represented by the following second-order term graph shown below, where solid, dashed and dotted lines represent the sets D, B and W respectively and reflexive relations are omitted.

[14] In fact, Bourbaki only makes the variable binding relation explicit, since he studies closed terms.



$$
\begin{aligned}
\mathcal{V} &= \{1,2,3,4,5,6\} \\
\mathit{label} &= \{(1,@),(2,\lambda),(3,\lambda),(4,@),(5,x),(6,x)\} \\
\mathit{succ} &= \{(1,[2,2]),(2,[3]),(3,[4]),(4,[5,6]),(5,[\,]),(6,[\,])\} \\
D &= \{(1,2),(2,3),(3,4),(4,5),(4,6)\} \\
B &= \{(5,2),(6,3)\} \\
W &= \{(5,5),(6,6)\}
\end{aligned}
$$

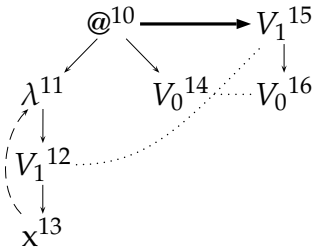
Kahl's formalism fully implements Klop's CRS, and so when used to implement the λ-calculus, the usual meta-substitution operation can be dispensed with. Recall that the β-substitution rule is:

$$(\beta): \qquad (\lambda x.M)N \rightarrow N\{M/x\}$$

When the rule is mapped and applied to a term, the meta-substitution operation expresses that the occurrences of the image of x in the image of M are replaced with the image of N. In the second-order syntax of CRS's, there is no need for this metasubstitution operation as the rule can be written directly as:

$$(\beta_2): (\lambda x.V_1(x))V_0 \rightarrow V_1(V_0)$$

Notice that the metavariable V_1 has direct access to occurrences of the bound variable x in its body. As a term graph in Kahl's system, the (β_2)-rule is specified as:



$$
\begin{aligned}
\mathcal{V} &= \{10,11,12,13,14,15,16\} \\
\mathit{label} &= \left\{ \begin{array}{l} (10,@),(11,\lambda),(12,V_1),(13,x),(14,V_0), \\ (15,V_1),(16,V_0) \end{array} \right\} \\
\mathit{succ} &= \left\{ \begin{array}{l} (10,[11,14]),(11,[12]),(12,[13]), \\ (13,[\,]),(15,[16]),(16,[\,]) \end{array} \right\} \\
D &= \{(10,11),(11,12),(12,13),(10,14),(15,16)\} \\
B &= \{(13,11)\} \\
W &= \left\{ \begin{array}{l} (12,15),(14,14),(14,16),(13,13),(12,12), \\ (15,15),(16,16) \end{array} \right\}
\end{aligned}
$$

The rewriting steps of Kahl's formalism are very involved, and so we will refer the reader to his original definitions [53, 52, 54]. Essentially the graph rewriting technique works by mapping each metavariable in the left-hand side of a rule to portions of the term graphs called *encapsulation skeletons*. These encapsulation skeletons are copied out in exactly the same way that Wadsworth copies the parts

of a λ-graphs that are not 'directly abstractable', thus maintaining the variable identity and variable binder relations.

The main observation of this section is that in all of these formalisms, during the reduction of a term, the binder relation needs to be maintained by some means. Usually, the capture avoiding substitution mechanism is computed to completion (i.e., eagerly) in an application of a rewrite rule. Explicit substitutions are an exception to this, since the propagation of the closure through the term structure depends on the evaluation strategy adopted.

2.5 Chapter Summary

In this chapter we have studied structural proof theory, computability theory and higher order rewriting. We summarise the important points made in these sections below.

Structural Proof Theory : Classical Logics are best implemented by sequent calculi, since these frameworks preserve the natural symmetries of the logic and natural permutations of proofs. The cut rule can be used to build concise proofs and is admissible in the logical system. The cut-elimination procedure is also *naturally* non-confluent and non-deterministic.

Logical connectives can be defined in sequent calculi in a proof-theoretic sense and/or a model-theoretic sense. For classical logics, entirely mechanical procedures exist that build sequent calculus rules from two-valued truth functions (represented as truth-tables). A *good* pair of logical introduction rules for a connective (in the proof-theoretic sense) has the property that whenever a logical connective is introduced as the cut-formula, that cut is eliminable from the proof. Various researchers have used (brute-force) resolution techniques to build the key *logical cut-elimination rule*. Since the permutability of proofs is not semantically important, obtaining more than one permutation of the logical cut-elimination rule *is not important* even though several may exist.

Computability Theory : Formal term calculi that more closely model the mechanisms of abstract machines have been incrementally developed. A strong relationship called the "Curry-Howard Isomorphism" exists between structural proof theory and computability theory (formulas of minimal implicative logic can be seen as types for λ-calculus terms).

Various researchers sought to extend this correspondence to a classical logic and sequent calculus setting. It has been found that the left and right introduction rules of a sequent calculus are interpreted as evaluation contexts and terms respectively, while cut-elimination corresponds to a notion of symmetric substitution.

The logical cut reduction rule for a connective represents the key computational rule for the term representation of the connective. (e.g. the logical cut reduction rule for implication corresponds to the plugging of an evaluation context with a term).

Lengrand has shown that the different permutations of the logical cut reduction rule *are* important: each permutation belongs to a different reduction subsystem. For the case of logical implication, there are two such permutations: one rule naturally belongs to a call-by-name subsystem, while the other belongs to a call-by-value system.

Higher Order Rewriting : Higher order languages feature notions of variable binding and variable identity that respectively relate a formal parameter to occurrences in the subterm and equate free variables. Such binding constructs give extra structure to a term which must be maintained during reduction (for reasons of correctness). Typically, the higher order languages also introduce a notion of term for variable substitution.

Various implementation techniques have been proposed that perform 'capture avoiding substitutions' (e.g. de Bruijn notation, Wadsworth's λ-graphs, Kahl's second-order term graphs, explicit substitutions). However, the key computational mechanism of the $\mathcal{X}$-calculus is not a term for variable substitution; rather it is a term for 'named-term' substitution, thus ruling out the direct applicability of many existing implementation techniques. We will introduce a new implementation technique in Section **??** that combines some features of those studied in the rewriting section.

Chapter 3

The (untyped) $\mathcal{X}$-Calculus

This motivational chapter studies the calculus of circuits $\mathcal{X}$ (pronounced "ex"), as first presented in [62] and studied in detail in [9]. We will present the formal definitions, and relate the syntax and reduction behaviour to the well-understood notions of computation studied in the background chapter (Section 2.3).

Although non-confluent, we will identify and study two confluent reduction subsystems that correspond to notions of call-by-name and call-by-value systems, and compare the non-confluence across the two systems. We will introduce generalise three aspects of the reduction system, leading to a more optimal reduction system.

3.1 Syntax and Reduction

In this section we will give the definition of the $\mathcal{X}$-calculus that was proven to be a fine-grained implementation model for various well-known calculi [9]. It features two separate categories of 'names', similar to the channel names used in the π-calculus; in $\mathcal{X}$ a name is a kind of *connector*: either a *plug* or a *socket*, which corresponds respectively to notions of output and input channels. In the following, we will us the words plugs and outputs interchangeably. Similarly, we will also use the symmetric notions of sockets inputs interchangeably.

Van Bakel, Lengrand and Lescanne [9] study the computational context of Classical Logic framed in a Sequent Calculus setting. They reformulate Urban's proof annotations (given by the syntax below), erase the types then study the corresponding calculus in an untyped setting.

Definition 3.1.1 ($\mathcal{X}$-Calculus Syntax) *The circuits of the $\mathcal{X}$-calculus are defined by the following grammar, where x, y range over the infinite set of* sockets*, and α, β over* plugs*.*

$$P, Q ::= \underset{\textit{capsule}}{\langle x \cdot \alpha \rangle} \mid \underset{\textit{export}}{\widehat{y} P \widehat{\beta} \cdot \alpha} \mid \underset{\textit{import}}{P \widehat{\beta}\, [y]\, \widehat{x} Q} \mid \underset{\textit{cut}}{P \widehat{\alpha} \dagger \widehat{x} Q}$$

The $\widehat{\cdot}$ symbolises that the socket (plug) underneath is bound in the circuit directly to the right (left). We will occasionally speak about the cut $\widehat{\alpha} \dagger \widehat{x}$ when speaking about the circuit $P\widehat{\alpha} \dagger \widehat{x}Q$ where we concentrate on the 'interface', rather than on the sub-circuits P and Q.

The novel syntax seeks to preserve the symmetries of computations at the level of the syntax of the calculus itself. The duality of (the computational notions of) input and output is reflected by the juxtaposition of the connector in relation to its sub-circuit: inputs (sockets) appear to the left of a sub-circuit while outputs (plugs) are on the right. Note that the import is syntactic sugar for $x \cdot [P\widehat{\beta}, \widehat{x}Q]$.

Unlike the usual term calculi used to model computations, there is no reserved symbol to represent the *binding relation* between a formal parameter and its occurrences in the subterm that is bound over. Returning full-circle, the authors re-adopt the original hat notation of Whitehead and Russell [76], where the variable under the hat identifies it as a formal parameter binding over the adjacent subterm[1]. (In Whitehead and Russell's syntax, the subterm was juxtaposed only to the right of the bound variable).

We give the definitions of free and bound connectors for $\mathcal{X}$-circuits below.

Definition 3.1.2 (Free and Bound Connectors) *The set of* free sockets *and* free plugs *in a circuit is defined by:*

$$\begin{array}{lll} fs(\langle x \cdot \alpha \rangle) & = \{x\} & fp(\langle x \cdot \alpha \rangle) = \{\alpha\} \\ fs(\widehat{y} P \widehat{\beta} \cdot \alpha) & = fs(P) \backslash \{y\} & fp(\widehat{x} P \widehat{\alpha} \cdot \beta) = (fp(P) \setminus \{\alpha\}) \cup \{\beta\} \\ fs(P \widehat{b}\, [y]\, \widehat{x} Q) & = fs(P) \cup (fs(Q) \backslash \{x\}) \cup \{y\} & fp(P \widehat{\alpha}\, [y]\, \widehat{x} Q) = (fp(P) \setminus \{\alpha\}) \cup fp(Q) \\ fs(P \widehat{\alpha} \dagger \widehat{x} Q) & = fs(P) \cup (fs(Q) \backslash \{x\}) & fp(P \widehat{\alpha} \dagger \widehat{x} Q) = (fp(P) \setminus \{\alpha\}) \cup fp(Q) \end{array}$$

The set of free connectors *is defined as $fc(P) = fs(P) \cup fp(P)$, and we sometimes write, for example, $fs(P, Q)$ as shorthand for $fs(P) \cup fs(Q)$.*

[1] "By the way, why did Church choose the notation 'λ'? [He] stated clearly that it came from the notation '$\widehat{x}$' used for class-abstraction by Whitehead and Russell, by first modifying '$\widehat{x}$' to '$\wedge x$' to distinguish function-abstraction from class-abstraction, and then changing '$\wedge$' to 'λ' for ease of printing ." [25, pp. 7]

The set of bound sockets *and* bound plugs *in a circuit is defined by:*

$$
\begin{array}{llll}
bs(\langle x{\cdot}\alpha\rangle) & = \emptyset & bp(\langle x{\cdot}\alpha\rangle) & = \emptyset \\
bs(\widehat{y}P\widehat{\beta}{\cdot}\alpha) & = bs(P) \cup \{y\} & bp(\widehat{x}P\widehat{\alpha}{\cdot}\beta) & = (bp(P) \cup \{\alpha\} \\
bs(P\widehat{\beta}\,[y]\,\widehat{x}Q) & = bs(P) \cup (bs(Q) \cup \{x\}) & bp(P\widehat{\alpha}\,[y]\,\widehat{x}Q) & = (bp(P) \cup \{\alpha\}) \cup bp(Q) \\
bs(P\widehat{\alpha} \dagger \widehat{x}Q) & = bs(P) \cup (bs(Q) \cup \{x\}) & bp(P\widehat{\alpha} \dagger \widehat{x}Q) & = (bp(P) \cup \{\alpha\}) \cup bp(Q)
\end{array}
$$

We use the notation $bc(P)$ $(= bs(P) \cup bp(P))$ for the bound connectors.

There are some non-standard features regarding the syntax of the $\mathcal{X}$-calculus. First notice that (with the exception of the capsule) each term constructor has exactly one free connector. Second, we note that some term constructors operate on more than one sub-circuit. This is in itself is not unusual; consider a function application term, or the closure $Q\langle x = P\rangle$ where x is a binder over Q. What is unusual in the $\mathcal{X}$-calculus syntax is that:

- There is no notion of abstraction; in particular, $P\widehat{\beta}$ and $\widehat{x}Q$ are not circuits; we will refer to them as *block*s.
- Some circuit constructors have several binders over the same sub-circuit.
- There are sometimes several sub-circuits that *each* have binders.

We will proceed by giving an intuitive description of how each term constructor can be understood computationally; we will refer to many of the concepts discussed in the background chapter (Section 2.3). We begin with a description of the capsule.

Capsule, $\langle x{\cdot}\alpha\rangle$: *Capsules* are the most basic term constructors and appear at the leaves of every $\mathcal{X}$ circuit. In $\mathcal{X}$, inputs are seen at the same level as outputs. This symmetry is reflected in the capsule by considering it to be either the input x that will send its result to the output named α, or, the output α expecting some input from x.

Import, $P\widehat{\beta}\,[y]\,\widehat{x}Q$: In applicative-style languages, a function consumer[2] might be written as $(\lambda x.Q)([\,]P)$, where the hole '$[\,]$' in the context is waiting for the insertion of a function which will consume the argument P; the continuation of this computation is the abstraction $\lambda x.Q$. Function consumers in continuation-style languages are able to separate the computational action of plugging a context from the construction of a function consumer, in the sense that the term representation of the function consumer is not a redex.

[2]We borrow terminology from Ariola *et al.* [1] that describes the computational behaviour of the term that corresponds to the sequent calculus left introduction rule for implication.

Such function consumers take the form of argument lists (like in $\overline{\lambda}\mu\tilde{\mu}$, where it is written $P \cdot Q$ where P is an argument and Q is the rest of the argument list).

The *import* in the $\mathcal{X}$-calculus is also a function consumer and can be thought of like an argument list in $\overline{\lambda}\mu\tilde{\mu}$. However, the syntax reflects (in our opinion) more closely what will happen during the interaction with the $\mathcal{X}$-circuit analog of function abstraction (the 'export'). P is a term from which an argument will be supplied (via the handle β) to the function consumer named y. The function supplied via y that takes the output provided by β will pass its own output (perhaps another function/export) to the rest of the argument list Q via x. In understanding this, the place of the supplied function really is *in between* the first argument and the rest of the argument list.

Export, $\widehat{y}P\widehat{\beta}\cdot\alpha$: An *export* is the $\mathcal{X}$-calculus analog of λ-abstraction. In the $\lambda\mu$-calculus, we could represent this as either $[\alpha]\lambda x.\mu\beta.P$ or $[\alpha]\mu\beta.\lambda x.P$, but notice that the $\mathcal{X}$ construct avoids this syntactic permutation. Familiarly, the bound socket x is the functional parameter and P is the body of the function. The bound plug β is a handle to the result of some computation in P. These two formal parameters work together as follows: when P is supplied an argument and continuation via the input and output channels, it will use the argument in its computation and send any result to the output channel β; this entire process is named α.

Cut, $P\widehat{\alpha} \dagger \widehat{x}Q$: Recall that the logical cut rule represents the computational notion of plugging the hole in an evaluation context with a term. In applicative-style calculi we may (loosely speaking) write the analog $(\lambda x.Q)P$, though this displays only part of the functionality. In $\overline{\lambda}\mu\tilde{\mu}$, we would write $\langle P \parallel Q \rangle$.

In the $\mathcal{X}$-calculus, the cut can be thought of as the plugging of the holes marked with x in Q with the arguments in P that output on α.

In the pure $\mathcal{X}$-calculus, we will identify circuits that differ only in the names of bound connectors (modulo α-conversion, as usual); in our implementation (of the tool) this is of course a problem to be dealt with (see Section 4.2).

The reduction rules of the $\mathcal{X}$-calculus describe in detail how cuts are propagated through circuits to be eventually evaluated at the level of capsules. The reduction rules are defined in two parts: (i) the *logical rules* describe the *direct action* of plugging the hole in an evaluation context with a term, (ii) the *propagation rules* specify how to transform the program so that the context and term are next to each other

and can directly interact. This criteria of being in the correct place before a direct interaction can take place strongly depends on the following notion.

Definition 3.1.3 (Introduction of connectors) *We define here what it means for a circuit to* introduce *a connector.*

P ***introduces*** x *: Either* $P = Q\widehat{\beta}\,[x]\,\widehat{y}R$ *and* $x \notin fs(Q,R)$*, or* $P = \langle x{\cdot}\alpha\rangle$
P ***introduces*** α *: Either* $P = \widehat{x}Q\widehat{\beta}{\cdot}\alpha$ *and* $\alpha \notin fp(Q)$*, or* $P = \langle x{\cdot}\alpha\rangle$

In the following, we may simply write x (α) is introduced *when it is clear which subcircuit the connector is introduced in.*

The direct action of plugging the hole of an evaluation context with a term is only possible when both connectors involved in the cut are *introduced*. In this case, computations are specified by the main reduction rules of the $\mathcal{X}$-calculus, given below.

Definition 3.1.4 (Logical rules) *The logical rules for the* $\mathcal{X}$*-calculus are presented below.*

$$
\begin{array}{rrcll}
(\textit{cap-rn}): & \langle y{\cdot}\alpha\rangle\widehat{\alpha}\,\dagger\,\widehat{x}\langle x{\cdot}\beta\rangle & \longrightarrow & \langle y{\cdot}\beta\rangle & \\
(\textit{exp-rn}): & (\widehat{y}P\widehat{\beta}{\cdot}\alpha)\widehat{\alpha}\,\dagger\,\widehat{x}\langle x{\cdot}\gamma\rangle & \longrightarrow & \widehat{y}P\widehat{\beta}{\cdot}\gamma & \longleftarrow\ \alpha \textit{ introduced} \\
(\textit{imp-rn}): & \langle y{\cdot}\alpha\rangle\widehat{\alpha}\,\dagger\,\widehat{x}(P\widehat{\beta}\,[x]\,\widehat{z}Q) & \longrightarrow & P\widehat{\beta}\,[y]\,\widehat{z}Q & \longleftarrow\ x \textit{ introduced} \\
(\textit{exp-imp}_{cbv}): & (\widehat{y}P\widehat{\beta}{\cdot}\alpha)\widehat{\alpha}\,\dagger\,\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R) & \longrightarrow & Q\widehat{\gamma}\,\dagger\,\widehat{y}(P\widehat{\beta}\,\dagger\,\widehat{z}R) & \longleftarrow\ \alpha, x \textit{ introduced} \\
(\textit{exp-imp}_{cbn}): & (\widehat{y}P\widehat{\beta}{\cdot}\alpha)\widehat{\alpha}\,\dagger\,\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R) & \longrightarrow & (Q\widehat{\gamma}\,\dagger\,\widehat{y}P)\widehat{\beta}\,\dagger\,\widehat{z}R & \longleftarrow\ \alpha, x \textit{ introduced}
\end{array}
$$

The first three logical rules above specify a renaming (reconnecting) procedure.

The last two rules are key computational rules describing the direct interaction between a 'function producer' and a 'function consumer' (borrowing the terminology of Ariola *et al.* [1]). The rules translate such an interaction into the language of plugging 'terms' and 'evaluation contexts'. We will study this pair of rules in more detail in Section 2.3.7 when we discuss confluent reduction subsystems in the $\mathcal{X}$-calculus.

These five rules only deal with those cases in which *both* connectors mentioned in the cut are introduced in their respective subterms. To define the propagation mechanism, we extend the syntax of the $\mathcal{X}$-calculus with two additional (and symmetric) term constructors. This move is motivated by Urban's strongly normalising cut elimination procedure [86] that annotates logical cuts with directions

that specify in which way the proof should be permuted so to shift the cut upwards through the structure of a derivation.

Definition 3.1.5 (Active cuts) *The syntax is extended with two* flagged *or* active *cuts:*

$$P, Q ::= \ldots \mid \underbrace{P\widehat{\alpha} \not\succ \widehat{x}Q}_{\textit{left-propagating cut}} \mid \underbrace{P\widehat{\alpha} \not\prec \widehat{x}Q}_{\textit{right-propagating cut}}$$

Terms constructed without these flagged cuts are called pure.

Cut elimination in the sequent calculus corresponds to a symmetrical notion of explicit substitutions (see Section 2.3.6). The dagger (†) used in the representation of the cut is tilted to indicate the direction the symmetric explicit substitution is propagating (this direction corresponds to the stabbing direction of the dagger).

To initiate the propagation mechanism, a cut is *activated* in a certain direction; the rules describing this step are given below.

Definition 3.1.6 (Activating) *We define two cut activation rules.*

$$(act\text{-}\mathrm{L}) : P\widehat{\alpha} \dagger \widehat{x}Q \longrightarrow P\widehat{\alpha} \not\succ \widehat{x}Q \leftarrow P \textit{ does not introduce } \alpha$$
$$(act\text{-}\mathrm{R}) : P\widehat{\alpha} \dagger \widehat{x}Q \longrightarrow P\widehat{\alpha} \not\prec \widehat{x}Q \leftarrow Q \textit{ does not introduce } x$$

In some situations, there is a choice to activate either to the left or to the right. Notice that then an additional source of non-determinism is created by the critical pair $(act\text{-}\mathrm{L})$ and $(act\text{-}\mathrm{R})$ and also leads to a highly non-confluent reduction mechanism. When we discuss reduction subsystems in Section 3.1.1, we will see how to recover the confluence of the calculus.

In the $\mathcal{X}$-calculus there are twelve symmetric propagation rules; these are not at all trivial to comprehend. In essence, the rules perform the task of 'pushing' a cut through the structure of a circuit while seeking out those connectors bound by the active cut (c.f. explicit substitutions). These rules are presented below.

Definition 3.1.7 (Propagation rules) *Right-propagation is reminiscent of* substitution *of terms for term-variables; left-propagation $P\widehat{\alpha} \not\succ \widehat{x}Q$ then is its dual: it expresses the connection of the continuations in Q, accessible via the handle x, to all subterms in P that output on α.*

Left propagation:

$$
\begin{array}{rll}
(\nleftarrow d): & \langle y\cdot\alpha\rangle\widehat{\alpha} \nleftarrow \widehat{x}P \to \langle y\cdot\alpha\rangle\widehat{\alpha} \dagger \widehat{x}P & \\
(cap\nleftarrow): & \langle y\cdot\beta\rangle\widehat{\alpha} \nleftarrow \widehat{x}P \to \langle y\cdot\beta\rangle & \leftarrow\ \beta \neq \alpha \\
(exp\text{-}outs\nleftarrow): & (\widehat{y}Q\widehat{\beta}\cdot\alpha)\widehat{\alpha} \nleftarrow \widehat{x}P \to (\widehat{y}(Q\widehat{\alpha} \nleftarrow \widehat{x}P)\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{x}P & \gamma \text{ fresh} \\
(exp\text{-}ins\nleftarrow): & (\widehat{y}Q\widehat{\beta}\cdot\gamma)\widehat{\alpha} \nleftarrow \widehat{x}P \to \widehat{y}(Q\widehat{\alpha} \nleftarrow \widehat{x}P)\widehat{\beta}\cdot\gamma & \leftarrow\ \gamma \neq \alpha \\
(imp\nleftarrow): & (Q\widehat{\beta}\,[z]\,\widehat{y}R)\widehat{\alpha} \nleftarrow \widehat{x}P \to (Q\widehat{\alpha} \nleftarrow \widehat{x}P)\widehat{\beta}\,[z]\,\widehat{y}(R\widehat{\alpha} \nleftarrow \widehat{x}P) & \\
(cut\nleftarrow): & (Q\widehat{\beta} \dagger \widehat{y}R)\widehat{\alpha} \nleftarrow \widehat{x}P \to (Q\widehat{\alpha} \nleftarrow \widehat{x}P)\widehat{\beta} \dagger \widehat{y}(R\widehat{\alpha} \nleftarrow \widehat{x}P) &
\end{array}
$$

Right propagation:

$$
\begin{array}{rll}
(d\nrightarrow): & P\widehat{\alpha} \nrightarrow \widehat{x}\langle x\cdot\beta\rangle \to P\widehat{\alpha} \dagger \widehat{x}\langle x\cdot\beta\rangle & \\
(\nrightarrow cap): & P\widehat{\alpha} \nrightarrow \widehat{x}\langle y\cdot\beta\rangle \to \langle y\cdot\beta\rangle & \leftarrow\ y \neq x \\
(\nrightarrow exp): & P\widehat{\alpha} \nrightarrow \widehat{x}(\widehat{y}Q\widehat{\beta}\cdot\gamma) \to \widehat{y}(P\widehat{\alpha} \nrightarrow \widehat{x}Q)\widehat{\beta}\cdot\gamma & \\
(\nrightarrow imp\text{-}outs): & P\widehat{\alpha} \nrightarrow \widehat{x}(Q\widehat{\beta}\,[x]\,\widehat{y}R) \to P\widehat{\alpha} \dagger \widehat{z}((P\widehat{\alpha} \nrightarrow \widehat{x}Q)\widehat{\beta}\,[z]\,\widehat{y}(P\widehat{\alpha} \nrightarrow \widehat{x}R)) & z \text{ fresh} \\
(\nrightarrow imp\text{-}ins): & P\widehat{\alpha} \nrightarrow \widehat{x}(Q\widehat{\beta}\,[z]\,\widehat{y}R) \to (P\widehat{\alpha} \nrightarrow \widehat{x}Q)\widehat{\beta}\,[z]\,\widehat{y}(P\widehat{\alpha} \nrightarrow \widehat{x}R) & \leftarrow\ z \neq x \\
(\nrightarrow cut): & P\widehat{\alpha} \nrightarrow \widehat{x}(Q\widehat{\beta} \dagger \widehat{y}R) \to (P\widehat{\alpha} \nrightarrow \widehat{x}Q)\widehat{\beta} \dagger \widehat{y}(P\widehat{\alpha} \nrightarrow \widehat{x}R) &
\end{array}
$$

We write

- $\to$ *for the (reflexive, transitive, compatible) reduction relation generated by the logical, propagation and activation rules.*
- $P \to^* Q$ *if there exists a reduction path from* P *to* Q.
- $P \downarrow Q$ *if* P *and* Q *have a common reduct, i.e., if there exists an* R *such that* $P \to R$ *and* $Q \to R$.
- $P \overset{\mathcal{X}}{=} Q$ *if* P *and* Q *have exactly the same normal forms.*

Also, we say an $\mathcal{X}$*-circuit is in* normal form *if it built without using the cut.*

We may now give a more detailed description of the propagation mechanism, referring to the above rules. An activated cut is processed by 'pushing' it systematically through the syntactic structure of the circuit in the direction indicated by the tilting of the dagger. The pushing of the active cut continues until the level of capsules is reached, where that cut is either deactivated or destroyed. Whenever an active cut meets a circuit exhibiting the connector it is trying to communicate with, a new (inactive) cut, with that connector made fresh, is 'deposited', representing an attempt to directly interact at this level. Once again, this new inactive cut can reduce via a logical rule, or pushing can continue in the other direction.

Notice that the rules $(\mathit{exp\text{-}outs}\not\nearrow)$ and $(\nwarrow\mathit{imp\text{-}outs})$ exemplify the creation of inactive cuts during propagation, as described above. For example, in the rule $(\nwarrow\mathit{imp\text{-}outs})$, the right-activated cut is pushed within the import, to have the connectors α and x link with each other in the sub-terms Q and R, and the cut '$\widehat{\alpha} \dagger \widehat{z}$' is placed outside to interact with the top socket. It is inactive since now the (fresh) connector z is now introduced, and perhaps a logical cut is applicable, or else activation in the other direction should take place.

In [9] some basic properties were shown, which essentially show that the calculus is well behaved.

Lemma 3.1.8 (Garbage Collection and Renaming [9]) *The following rules apply, where in each case the sub-circuit P is* pure.

$$(\not\nearrow gc_p) : P\widehat{\alpha} \not\nearrow \widehat{x}Q \longrightarrow P, \twoheadleftarrow\ \alpha \notin fp(P) \qquad (\mathit{ren}\text{-}\mathrm{L}_p) :\ P\widehat{\delta} \dagger \widehat{z}\langle z{\cdot}\alpha\rangle, \longrightarrow P\{\alpha/\delta\}$$
$$(\nwarrow gc_p) : Q\widehat{\alpha} \nwarrow \widehat{x}P \longrightarrow P, \twoheadleftarrow\ x \notin fs(P) \qquad (\mathit{ren}\text{-}\mathrm{R}_p) : \langle z{\cdot}\alpha\rangle\widehat{\alpha} \dagger \widehat{x}P, \longrightarrow P\{z/x\}$$

NB: in [9], these results were shown for *pure* P only, a restriction we will drop here (see Lemma 3.1.21 and 3.1.23).

To end this section, we would like to comment on the overall effect of the reduction system. In Bloo and Rose's λx-calculus (reviewed in Section 2.4.2), the overall effect of the reduction relation is to perform a 'term for variable' substitution operation where a copy of the term is simply slotted in place of each occurrence of the variable in the subterm (i.e., of the shape $P\{Q/x\}$). In the $\mathcal{X}$-calculus, this notion only applies to the renaming and garbage collection rules given above. In the general case, the 'substitution' operation (which we recall is also symmetrical) is performed quite differently. Unfortunately, there is no standard notion to express this, though Urban [86] and Summers [84] have defined their own notations. For the cut $P\widehat{\alpha} \dagger \widehat{x}Q$, the shape of Urban's 'substitution' construct is like[3]:

$$P[\alpha := (x)Q] \quad \textit{and} \quad Q[x := \langle\alpha\rangle P]$$

And Summers writes:

$$P\{\alpha \leftrightarrow \widehat{x}Q\} \quad \textit{and} \quad Q\{P\widehat{\alpha} \leftrightarrow x\}$$

These constructs internalise the explicit propagation rules (Definition 3.1.7). The

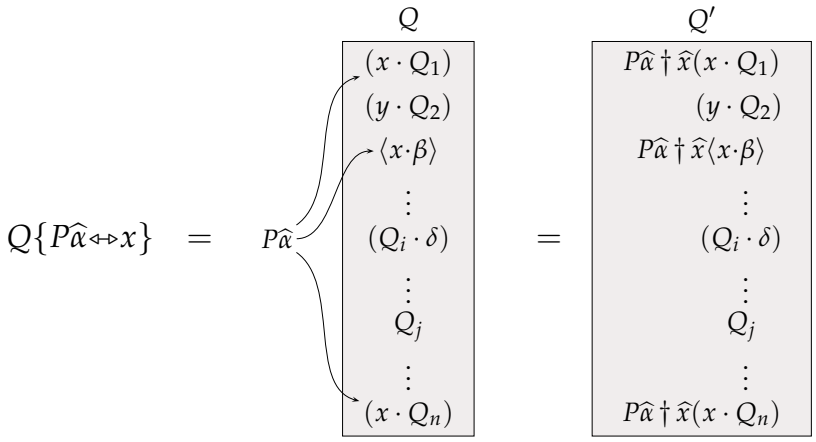
[3]We adopt the two alphabets (Roman and Greek) of the $\mathcal{X}$-calculus to make the comparison more clear.

left construct attempts to reflect the fact that the *block* $\widehat{x}Q$ will be copied and *connected to* each of the α's in P. Symmetrically, the right construct reflects that the *block* $P\widehat{\alpha}$ will be copied to sub-circuits exhibiting x in Q. Unlike the usual notion of substitution, the terms are not simply slotted into the places marked with x or α. Rather, a new cut is formed with the respective sub-circuits. Taking $Q\{P\widehat{\alpha} \leftrightarrow x\}$ as an example, suppose we list (in some deterministic order) all the sub-circuits of Q,

$$(x \cdot Q_1), (y \cdot Q_2), \langle x \cdot \beta \rangle, \ldots, (Q_i \cdot \delta), \ldots, Q_j, \ldots, (x \cdot Q_n).$$

Here we allow for some leniency in notation, where the Q_k (for $0 < k \leq n$) terms are lists of *block*s; we write $z \cdot Q_k$ to represent the sub-circuit that has the connector z *free* at its top-level.

Then we can illustrate the 'substitution-like' process as follows,



Each of the newly formed cuts in the sub-circuit Q' *introduces* the socket x.

3.1.1 Reduction Subsystems for $\mathcal{X}$

The reduction relation of the $\mathcal{X}$-calculus, $\rightarrow$, is not confluent; this comes in fact from the critical pair that activates a cut $P\widehat{\alpha} \dagger \widehat{x}Q$ in two ways if P does not contain (so does not introduce) α and Q does not contain x. In his case, we have both $P\widehat{\alpha} \dagger \widehat{x}Q \rightarrow P$ and $P\widehat{\alpha} \dagger \widehat{x}Q \rightarrow Q$.

Lengrand in [61, 62] defines two subsystems of his $\lambda\xi$-calculus that correspond to call-by-name (CBN) and call-by-value (CBV) notions of reduction. These were carried over to the $\mathcal{X}$-calculus in [9]; the two subsystems of reduction are defined below.

Definition 3.1.9 (Call-by-Value Subsystem) *We place two restrictions on the pure $\mathcal{X}$-calculus presented in Section 3.1 to obtain a confluent call-by-value subsystem.*

First, if a cut can be activated in two ways, the CBV *subsystem only allows to activate it via* $(act\text{-}\mathrm{L})$*. We can reformulate this as the reduction system obtained by replacing rule* $(act\text{-}\mathrm{R})$ *by:*

$$(act\text{-}\mathrm{R}) : P\widehat{\alpha} \dagger \widehat{x}Q \longrightarrow P\widehat{\alpha} \nwarrow \widehat{x}Q \longleftarrow P \text{ introduces } \alpha \text{ and } Q \text{ does not introduce } x$$

Secondly, we remove *the rule* $(exp\text{-}imp_{cbn})$ *from the set of reduction rules, leaving only the* CBV *variant.*

$$(exp\text{-}imp_{cbv}) : (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R) \longrightarrow Q\widehat{\gamma} \dagger \widehat{y}(P\widehat{\beta} \dagger \widehat{z}R) \longleftarrow \alpha, x \text{ introduced}$$

We will write $P \longrightarrow_{\mathrm{V}} Q$ *to represent call-by-value reductions.*

Definition 3.1.10 (Call-by-Name Subsystem) *To obtain a confluent call-by-name reduction subsystem, we place the following two restrictions on the pure $\mathcal{X}$-calculus (as presented in Section 3.1).*

First, if a cut can be activated in two ways, the CBN *subsystem only allows to activate it via* $(act\text{-}\mathrm{R})$ *. We can reformulate this as the reduction system obtained by replacing rule* $(act\text{-}\mathrm{L})$ *by:*

$$(act\text{-}\mathrm{L}) : P\widehat{\alpha} \dagger \widehat{x}Q \longrightarrow P\widehat{\alpha} \nearrow \widehat{x}Q \longleftarrow Q \text{ introduces } x \text{ and } P \text{ does not introduce } \alpha$$

In addition, we remove *the rule* $(exp\text{-}imp_{cbv})$ *from the reduction relation, leaving only the* CBN *variant.*

$$(exp\text{-}imp_{cbn}) : (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R) \longrightarrow (Q\widehat{\gamma} \dagger \widehat{y}P)\widehat{\beta} \dagger \widehat{z}R \longleftarrow \alpha, x \text{ introduced}$$

We will write $P \longrightarrow_{\mathrm{N}} Q$ *for reductions done under the call-by-name subsystem.*

Observe that the rule $(exp\text{-}imp_{cbv})$ is structured so that preference is given to supplying the continuations named z in R to the terms in P that output on β, while the $(exp\text{-}imp_{cbn})$ appears to prefer supplying the arguments that outputs on γ in Q to the function P via its parameter y. The two systems defined above correspond to Herbelin's notions of call-by-name and call-by-value reductions as discussed in Section 2.3.7.

The two restriction on the reduction relation automatically give confluent subcalculi since the all rules are left-linear and non-overlapping.

3.1.2 $\mathcal{X}$ as a General Reduction Machine

In [9] the relation between $\mathcal{X}$ and many other calculi is studied; as an illustration, in this section, we will briefly highlight the relation between the λ-calculus and $\mathcal{X}$. (We reviewed the λ-calculus in Section 2.3.1).

Due to the Curry-Howard correspondence between the λ-calculus and the natural deduction formulation of minimal implicative logic on the one hand, and between the $\mathcal{X}$-calculus and the sequent calculus formulation of classical logic on the other hand, existing encodings that translate natural deduction proofs to sequent calculus proofs can be used to obtain an encoding from the λ-calculus to the $\mathcal{X}$-calculus. There are two different well-studied encodings, defined respectively by Gentzen [41] and Prawitz [71]. While Gentzen's version is more straightforward, Prawitz's version preserves the status of normal forms in an encoding, that is, a normalised natural deduction proof translates to a cut-free sequent proof. We give the two corresponding encodings that translate λ-terms to $\mathcal{X}$-terms below.

Definition 3.1.11 (Interpreting the λ-calculus *à la* Gentzen [41, 9]) *The interpretation of λ-terms into circuits of $\mathcal{X}$ in the context α, $\lfloor M \rfloor_\alpha$, is defined by:*

$$\begin{aligned}
\lfloor x \rfloor_\alpha &= \langle x \cdot \alpha \rangle \\
\lfloor \lambda x.M \rfloor_\alpha &= \widehat{x} \lfloor M \rfloor_\beta \widehat{\beta} \cdot \alpha && \beta \text{ fresh} \\
\lfloor MN \rfloor_\alpha &= \lfloor M \rfloor_\gamma \widehat{\gamma} \dagger \widehat{x} (\lfloor N \rfloor_\beta \widehat{\beta} \, [x] \, \widehat{y} \langle y \cdot \alpha \rangle) && x, y, \beta, \gamma \text{ fresh}
\end{aligned}$$

We can even represent substitution explicitly (so represent $\lambda\boldsymbol{x}$, Section 2.4.2), by adding

$$\lfloor M \langle x = N \rangle \rfloor_\alpha = \lfloor N \rfloor_\gamma \widehat{\gamma} \nwarrow \widehat{x} \lfloor M \rfloor_\alpha \qquad \gamma \text{ fresh}$$

Definition 3.1.12 (Interpreting the λ-calculus *à la* Prawitz [71, 82]) *There are three parts to the interpretation. The symbols L are variables ranging over lists of λ-terms, and*

the symbols M, N, P to represent arbitrary λ-terms.

$$
\begin{array}{lll}
a) & \ulcorner x \urcorner_\alpha & = \langle x \cdot \alpha \rangle \\
& \ulcorner \lambda x.M \urcorner_\alpha & = \widehat{x} \ulcorner M \urcorner_\beta \widehat{\beta} \cdot \alpha \\
& \ulcorner (MN) \urcorner_\alpha & = \ulcorner (MN), [\,] \urcorner_\alpha \\
\\
b) & \ulcorner (xN), L \urcorner_\alpha & = \ulcorner N \urcorner_\beta \widehat{\beta}\, [x]\, \widehat{y} \ulcorner L \urcorner_\alpha^y \\
& \ulcorner (\lambda x.M)N, L \urcorner_\alpha & = \ulcorner \lambda x.M \urcorner_\beta \widehat{\beta} \dagger \widehat{y} (\ulcorner N \urcorner_\gamma \widehat{\gamma}\, [y]\, \widehat{z} \ulcorner L \urcorner_\alpha^z) \\
& \ulcorner (MN)P, L \urcorner_\alpha & = \ulcorner (MN), P : L \urcorner_\alpha \\
\\
c) & \ulcorner [\,] \urcorner_\alpha^x & = \langle x \cdot \alpha \rangle \\
& \ulcorner M : L \urcorner_\alpha^x & = \ulcorner M \urcorner_\beta \widehat{\beta}\, [x]\, \widehat{y} \ulcorner L \urcorner_\alpha^y
\end{array}
$$

Under both interpretations, every interpreted λ-subterm has exactly one free plug; this is easiest to see in Gentzen's translation. This observation motivates the reason why there is no explicit notion of output in the λ-calculus: each term has exactly one output, and the juxtaposition of terms eliminates the possible ambiguity of where a term might send its result.

In [9], the following relation is shown between (call-by-name, call-by-value) reduction in λ-calculus and $\mathcal{X}$:

Theorem 3.1.13 ([9]) *The following reduction properties have been shown to hold for Gentzen's interpretation of λ-calculus terms to $\mathcal{X}$.*

1. *If $M \to_\beta N$ then $\lfloor M \rfloor_\alpha \to \lfloor N \rfloor_\alpha$*
2. *If $M \to_{\mathrm{V}} N$ then $\lfloor M \rfloor_\alpha \to_{\mathrm{V}} \lfloor N \rfloor_\alpha$*
3. *If $M \to_{\mathrm{N}} N$ then $\lfloor M \rfloor_\alpha \to_{\mathrm{N}} \lfloor N \rfloor_\alpha$*

As a matter of fact, the last two results link the concept of '*name*' and '*value*' quite nicely to $\mathcal{X}$: the circuits that can be called a *value* in $\mathcal{X}$ are those that introduce a plug, and a *name* is a circuit that introduces a socket. However, notice that, in contrast to the λ-calculus, in $\mathcal{X}$ the CBV reduction is not a sub-subsystem of the CBN reduction. For the CBV reduction on $\mathcal{X}$, the cut $P\widehat{\alpha} \dagger \widehat{x}Q$ is only right-activated if Q does not introduce x, and P is a value. So, P is only 'inserted' into Q if it is a value, which makes this reduction justifiably called 'call-by-value'.

The converse of the results of Theorem 3.1.13 do not hold a-priori: this is mainly because the reduction relation in $\mathcal{X}$ is far more complex than just those reductions

between (interpretations of) λ-terms, and it could be that there exists a path between $\lfloor M \rfloor_\alpha$ and $\lfloor N \rfloor_\alpha$ which does not correspond to a λ-calculus-reduction path between M and N.

It is worthwhile to notice that the image of the set of λ-terms under the interpretation function $\lfloor \cdot \rfloor_\alpha$ does not generate a confluent sub-calculus. We illustrate this by the following:

Example 3.1.14 (Non-confluence across CBV and CBN, [9]) *In the following, we will make use of* $\lfloor xx \rfloor_\alpha \to \langle x \cdot \beta \rangle \widehat{\beta}\,[x]\,\widehat{y}\langle y \cdot \alpha \rangle$.

$$
\begin{array}{lll}
\lfloor (\lambda x.xx)(yy) \rfloor_\alpha & \triangleq & \\
\lfloor \lambda x.xx \rfloor_\beta \widehat{\beta} \dagger \widehat{v}(\lfloor yy \rfloor_\gamma \widehat{\gamma}\,[v]\,\widehat{w}\langle w \cdot \alpha \rangle) & \triangleq & \\
(\widehat{x}\lfloor xx \rfloor_\delta \widehat{\delta} \cdot \beta)\widehat{\beta} \dagger \widehat{v}(\lfloor yy \rfloor_\gamma \widehat{\gamma}\,[v]\,\widehat{w}\langle w \cdot \alpha \rangle) & \to & (exp\text{-}imp) \\
\lfloor yy \rfloor_\gamma \widehat{\gamma} \dagger \widehat{x}(\lfloor xx \rfloor_\delta \widehat{\delta} \dagger \widehat{w}\langle w \cdot \alpha \rangle) & \to & (act\text{-}\mathrm{L}), (ren\text{-}\mathrm{R}) \\
\lfloor yy \rfloor_\gamma \widehat{\gamma} \dagger \widehat{x}\lfloor xx \rfloor_\alpha & \to^* & \\
(\langle y \cdot \sigma \rangle \widehat{\sigma}\,[y]\,\widehat{z}\langle z \cdot \gamma \rangle)\widehat{\gamma} \dagger \widehat{x}(\langle x \cdot \tau \rangle \widehat{\tau}\,[x]\,\widehat{u}\langle u \cdot \alpha \rangle) & &
\end{array}
$$

This circuit now has one cut only, that can be activated in two ways (notice that neither γ *nor* x *is introduced here). Under* CBV*, this results in:*

$$
\begin{array}{lll}
\lfloor yy \rfloor_\gamma \widehat{\gamma} \dagger \widehat{x}\lfloor xx \rfloor_\alpha & \to & (act\text{-}\mathrm{L}) \\
\lfloor yy \rfloor_\gamma \widehat{\gamma} \nleftarrow \widehat{x}\lfloor xx \rfloor_\alpha & \to & \\
(\langle y \cdot \sigma \rangle \widehat{\sigma}\,[y]\,\widehat{z}\langle z \cdot \gamma \rangle)\widehat{\gamma} \nleftarrow \widehat{x}\lfloor xx \rfloor_\alpha & \to & (imp\nleftarrow) \\
(\langle y \cdot \sigma \rangle \widehat{\gamma} \nleftarrow \widehat{x}\lfloor xx \rfloor_\alpha)\widehat{\sigma}\,[y]\,\widehat{z}(\langle z \cdot \gamma \rangle \widehat{\gamma} \nleftarrow \widehat{x}\lfloor xx \rfloor_\alpha) & \to & (cap\nleftarrow), (\nleftarrow d), (act\text{-}\mathrm{R}) \\
\langle y \cdot \sigma \rangle \widehat{\sigma}\,[y]\,\widehat{z}(\langle z \cdot \gamma \rangle \widehat{\gamma} \nwarrow \widehat{x}\lfloor xx \rfloor_\alpha) & \to & (ren\text{-}\mathrm{L}) \\
\langle y \cdot \sigma \rangle \widehat{\sigma}\,[y]\,\widehat{z}(\langle z \cdot \tau \rangle \widehat{\tau}\,[z]\,\widehat{u}\langle u \cdot \alpha \rangle) & &
\end{array}
$$

or under CBN*:*

$$
\begin{array}{lll}
\lfloor yy \rfloor_\gamma \widehat{\gamma} \dagger \widehat{x}\lfloor xx \rfloor_\alpha & \to & \\
\lfloor yy \rfloor_\gamma \widehat{\gamma} \dagger \widehat{x}(\langle x \cdot \tau \rangle \widehat{\tau}\,[x]\,\widehat{u}\langle u \cdot \alpha \rangle) & \to & (act\text{-}\mathrm{R}), (\nwarrow imp\text{-}outs) \\
\lfloor yy \rfloor_\gamma \widehat{\gamma} \dagger \widehat{w}((\lfloor yy \rfloor_\gamma \widehat{\gamma} \nwarrow \widehat{x}\langle x \cdot \tau \rangle)\widehat{\tau}\,[w]\,\widehat{u}(\lfloor yy \rfloor_\gamma \widehat{\gamma} \nwarrow \widehat{x}\langle u \cdot \alpha \rangle)) & \to & (d\nwarrow), (act\text{-}\mathrm{L}), (\nwarrow cap) \\
\lfloor yy \rfloor_\gamma \widehat{\gamma} \dagger \widehat{w}((\lfloor yy \rfloor_\gamma \widehat{\gamma} \nleftarrow \widehat{x}\langle x \cdot \tau \rangle)\widehat{\tau}\,[w]\,\widehat{u}\langle u \cdot \alpha \rangle) & \to & (ren\text{-}\mathrm{R}) \\
\lfloor yy \rfloor_\gamma \widehat{\gamma} \nleftarrow \widehat{w}(\lfloor yy \rfloor_\tau \widehat{\tau}\,[w]\,\widehat{u}\langle u \cdot \alpha \rangle) & \to & \\
(\langle y \cdot \sigma \rangle \widehat{\sigma}\,[y]\,\widehat{z}\langle z \cdot \gamma \rangle)\widehat{\gamma} \nleftarrow \widehat{w}(\lfloor yy \rfloor_\tau \widehat{\tau}\,[w]\,\widehat{u}\langle u \cdot \alpha \rangle) & \to & (imp\nleftarrow) \\
(\langle y \cdot \sigma \rangle \widehat{\gamma} \nleftarrow \widehat{w}(\lfloor yy \rfloor_\tau \widehat{\tau}\,[w]\,\widehat{u}\langle u \cdot \alpha \rangle))\widehat{\sigma}\,[y] & & \\
\qquad \widehat{z}(\langle z \cdot \gamma \rangle \widehat{\gamma} \nleftarrow \widehat{w}(\lfloor yy \rfloor_\tau \widehat{\tau}\,[w]\,\widehat{u}\langle u \cdot \alpha \rangle)) & \to & (cap\nleftarrow), (\nleftarrow d), \\
\langle y \cdot \sigma \rangle \widehat{\sigma}\,[y]\,\widehat{z}(\langle z \cdot \gamma \rangle \widehat{\gamma} \dagger \widehat{w}(\lfloor yy \rfloor_\tau \widehat{\tau}\,[w]\,\widehat{u}\langle u \cdot \alpha \rangle)) & \to & (imp\text{-}rn) \\
\langle y \cdot \sigma \rangle \widehat{\sigma}\,[y]\,\widehat{z}((\langle y \cdot \sigma \rangle \widehat{\sigma}\,[y]\,\widehat{z}\langle z \cdot \tau \rangle)\widehat{\tau}\,[z]\,\widehat{u}\langle u \cdot \alpha \rangle) & &
\end{array}
$$

Notice that both reductions return *different* normal forms, so $\lfloor(\lambda x.xx)(yy)\rfloor_\alpha$ has *two* normal forms. Even though $(\lambda x.xx)(yy)$ also has two different normal forms with respect to CBV and CBN reduction (respectively $(\lambda x.xx)(yy)$ and $(yy)(yy)$), the structures of the normal forms obtained here are of a different nature. We will illustrate this with another example.

Example 3.1.15 (A comparison of a CBV and CBN reduction) *The computational behaviour of the normal forms obtained in Example 3.1.14 can be compared to computational machines that behave (respectively) like 'serial' and 'parallel' function consumers.*

First observe that both normal forms are indeed argument lists (or 'function consumers'). The difference is, the CBV *argument list processes the function using a staggered or step-by-step approach, while the* CBN *argument list eagerly processes the function by passing it through its entire list. We will illustrate this behaviour by considering its 'application' to the $\mathcal{X}$-calculus analog of the identity, $\lfloor\lambda p.p\rfloor_\varrho = \widehat{p}\langle p\cdot\circ\rangle\widehat{\circ}\cdot\varrho$.*

For the CBV *reduction, we have:*

$$
\begin{array}{ll}
 & \lfloor\lambda p.p\rfloor_\varrho\widehat{\varrho}\dagger\widehat{y}(\langle y\cdot\sigma\rangle\widehat{\sigma}\,[y]\,\widehat{z}(\langle z\cdot\tau\rangle\widehat{\tau}\,[z]\,\widehat{u}\langle u\cdot\alpha\rangle)) \\
\to & \lfloor\lambda p.p\rfloor_\varrho\widehat{\varrho}\nwarrow\widehat{y}(\langle y\cdot\sigma\rangle\widehat{\sigma}\,[y]\,\widehat{z}(\langle z\cdot\tau\rangle\widehat{\tau}\,[z]\,\widehat{u}\langle u\cdot\alpha\rangle)) \\
\to^* & \lfloor\boldsymbol{\lambda p.p}\rfloor_\varrho\widehat{\varrho}\dagger\widehat{k}(\lfloor\boldsymbol{\lambda p.p}\rfloor_\sigma\widehat{\sigma}\,[k]\,\widehat{z}(\langle z\cdot\tau\rangle\widehat{\tau}\,[z]\,\widehat{u}\langle u\cdot\alpha\rangle)) \\
= & (\widehat{p}\langle p\cdot\circ\rangle\widehat{\circ}\cdot\varrho)\widehat{\varrho}\dagger\widehat{k}(\lfloor\lambda p.p\rfloor_\sigma\widehat{\sigma}\,[k]\,\widehat{z}(\langle z\cdot\tau\rangle\widehat{\tau}\,[z]\,\widehat{u}\langle u\cdot\alpha\rangle)) \\
\to^* & \lfloor\lambda p.p\rfloor_\sigma\widehat{\sigma}\dagger\widehat{p}(\langle p\cdot\circ\rangle\widehat{\circ}\dagger\widehat{z}(\langle z\cdot\tau\rangle\widehat{\tau}\,[z]\,\widehat{u}\langle u\cdot\alpha\rangle)) \\
\to^* & \lfloor\lambda p.p\rfloor_\sigma\widehat{\sigma}\dagger\widehat{p}(\langle p\cdot\tau\rangle\widehat{\tau}\,[p]\,\widehat{u}\langle u\cdot\alpha\rangle) \\
\to^* & \lfloor\boldsymbol{\lambda p.p}\rfloor_\sigma\widehat{\sigma}\dagger\widehat{c}(\lfloor\boldsymbol{\lambda p.p}\rfloor_\tau\widehat{\tau}\,[c]\,\widehat{u}\langle u\cdot\alpha\rangle) \\
= & (\widehat{p}\langle p\cdot\circ\rangle\widehat{\circ}\cdot\sigma)\widehat{\sigma}\dagger\widehat{c}(\lfloor\lambda p.p\rfloor_\tau\widehat{\tau}\,[c]\,\widehat{u}\langle u\cdot\alpha\rangle) \\
\to & (\lfloor\lambda p.p\rfloor_\tau\widehat{\tau}\dagger\widehat{p}\langle p\cdot\circ\rangle)\widehat{\circ}\dagger\widehat{u}\langle u\cdot\alpha\rangle \\
\to & \lfloor\lambda p.p\rfloor_\circ\widehat{\circ}\dagger\widehat{u}\langle u\cdot\alpha\rangle \\
\to & \lfloor\lambda p.p\rfloor_\alpha
\end{array}
$$

For the CBN *reduction, we have:*

$$
\begin{aligned}
& \lfloor \lambda p.p \rfloor_\varrho \widehat{\varrho} \dagger \widehat{y}(\langle y{\cdot}\sigma \rangle \widehat{\sigma}\,[y]\,\widehat{z}(\langle z{\cdot}\tau \rangle)\widehat{\tau}\,[z]\,\widehat{u}\langle u{\cdot}\alpha \rangle)) \\
\to\ & \lfloor \lambda p.p \rfloor_\varrho \widehat{\varrho} \nwarrow \widehat{y}(\langle y{\cdot}\sigma \rangle \widehat{\sigma}\,[y]\,\widehat{z}(\langle z{\cdot}\tau \rangle)\widehat{\tau}\,[z]\,\widehat{u}\langle u{\cdot}\alpha \rangle)) \\
\to^*\ & \lfloor \boldsymbol{\lambda p.p} \rfloor_\varrho \widehat{\varrho} \dagger \widehat{k}(\lfloor \boldsymbol{\lambda p.p} \rfloor_\sigma \widehat{\sigma}\,[k]\,\widehat{z}(\langle \lfloor \boldsymbol{\lambda p.p} \rfloor_\varrho \widehat{\varrho} \dagger \widehat{h}(\lfloor \boldsymbol{\lambda p.p} \rfloor_\sigma \widehat{\sigma}\,[h]\,\widehat{z}\langle z{\cdot}\tau \rangle))\widehat{\tau}\,[z]\,\widehat{u}\langle u{\cdot}\alpha \rangle)) \\
\to^*\ & (\lfloor \lambda p.p \rfloor_\sigma \widehat{\sigma} \dagger \widehat{p}\langle p{\cdot}\circ \rangle)\widehat{\circ} \dagger \widehat{z}(\langle \lfloor \lambda p.p \rfloor_\varrho \widehat{\varrho} \dagger \widehat{h}(\lfloor \lambda p.p \rfloor_\sigma \widehat{\sigma}\,[h]\,\widehat{z}\langle z{\cdot}\tau \rangle))\widehat{\tau}\,[z]\,\widehat{u}\langle u{\cdot}\alpha \rangle) \\
\to^*\ & \lfloor \lambda p.p \rfloor_\circ \widehat{\circ} \dagger \widehat{z}(\langle \lfloor \lambda p.p \rfloor_\varrho \widehat{\varrho} \dagger \widehat{h}(\lfloor \lambda p.p \rfloor_\sigma \widehat{\sigma}\,[h]\,\widehat{z}\langle z{\cdot}\tau \rangle))\widehat{\tau}\,[z]\,\widehat{u}\langle u{\cdot}\alpha \rangle) \\
\to^*\ & (\langle \lfloor \lambda p.p \rfloor_\varrho \widehat{\varrho} \dagger \widehat{h}(\lfloor \lambda p.p \rfloor_\sigma \widehat{\sigma}\,[h]\,\widehat{z}\langle z{\cdot}\tau \rangle))\widehat{\tau} \dagger \widehat{p}\langle p{\cdot}\circ \rangle)\widehat{\circ} \dagger \widehat{u}\langle u{\cdot}\alpha \rangle \\
\to^*\ & (\lfloor \lambda p.p \rfloor_\varrho \widehat{\varrho} \dagger \widehat{h}(\lfloor \lambda p.p \rfloor_\sigma \widehat{\sigma}\,[h]\,\widehat{z}(\langle z{\cdot}\tau \rangle \widehat{\tau} \dagger \widehat{p}\langle p{\cdot}\circ \rangle)))\widehat{\circ} \dagger \widehat{u}\langle u{\cdot}\alpha \rangle \\
\to^*\ & (\lfloor \lambda p.p \rfloor_\varrho \widehat{\varrho} \dagger \widehat{h}(\lfloor \lambda p.p \rfloor_\sigma \widehat{\sigma}\,[h]\,\widehat{z}\langle z{\cdot}\circ \rangle))\widehat{\circ} \dagger \widehat{u}\langle u{\cdot}\alpha \rangle \\
\to^*\ & \lfloor \lambda p.p \rfloor_\varrho \widehat{\varrho} \dagger \widehat{h}(\lfloor \lambda p.p \rfloor_\sigma \widehat{\sigma}\,[h]\,\widehat{z}\langle z{\cdot}\alpha \rangle) \\
\to^*\ & (\lfloor \lambda p.p \rfloor_\sigma \widehat{\sigma} \dagger \widehat{p}\langle p{\cdot}\circ \rangle)\widehat{\circ} \dagger \widehat{z}\langle z{\cdot}\alpha \rangle \\
\to^*\ & \lfloor \lambda p.p \rfloor_\circ \widehat{\circ} \dagger \widehat{z}\langle z{\cdot}\alpha \rangle \\
\to\ & \lfloor \lambda p.p \rfloor_\alpha
\end{aligned}
$$

We appreciate the reader unfamiliar with the $\mathcal{X}$-calculus will find these terms difficult to parse, but we would like to highlight the copying of the function $\lfloor \lambda p.p \rfloor_\varrho$. The CBV *function consumer proceeds by allowing the function to be applied to the first argument, and only when that argument has been evaluated can the next argument from the list be supplied.*

In contrast, the CBN *structure eagerly applies the function to all argument of the argument list first. This is illustrated by the four identity circuits highlighted in bold.*

Aside from structural behaviour of argument lists, we would like to give some intuition behind how some more practical computations are performed in the $\mathcal{X}$-calculus. We will therefore study the interpretation of the program Example 2.3.9 from the background section in the context of $\mathcal{X}$.

Example 3.1.16 (A Reduction in $\mathcal{X}$) *In the $\mathcal{X}$-calculus (compared to the $\overline{\lambda}\mu\tilde{\mu}$-calculus) the result of every computation is explicitly 'named', meaning there is a syntactic representation for the continuation of every computation. For example, if we wish to say the result of computing $(x{+}y)$ is passed to the output named δ, like $\delta{=}(x{+}y)$, we can express this using the capsule, i.e., $\langle (x{+}y){\cdot}\delta \rangle$.*

Using Prawitz's normal-form preserving translation of Definition 3.1.12, we can interpret the λ-calculus program of Example 2.3.9 to $\mathcal{X}$ enriched with natural numbers and arithmetic operations; this is shown below.

$$
\begin{aligned}
\lceil add \rceil_\varphi &= \widehat{u}(\widehat{v}\langle (u{+}v){\cdot}\chi \rangle \widehat{\chi}{\cdot}v)\widehat{v}{\cdot}\varphi \\
\lceil divideTwo \rceil_\nu &= \widehat{a}\langle (a/2){\cdot}\pi \rangle \widehat{\pi}{\cdot}\nu \\
\lceil average \rceil_\delta &= \ldots
\end{aligned}
$$

$$
\widehat{x}(\widehat{y}(\lceil divideTwo \rceil_\nu \widehat{\nu} \dagger \widehat{c}(\langle \lceil add \rceil_\varphi \widehat{\varphi} \dagger \widehat{i}(\langle \boldsymbol{y}{\cdot}\circ \rangle \widehat{\circ}\,[i]\,\widehat{t}(\langle \boldsymbol{x}{\cdot}\lambda \rangle \widehat{\lambda}\,[t]\,\widehat{s}\langle s{\cdot}\gamma \rangle)))\widehat{\gamma}\,[c]\,\widehat{p}\langle p{\cdot}\sigma \rangle))\widehat{\sigma}{\cdot}\tau)\widehat{\tau}{\cdot}\delta
$$

The $\lceil average \rceil_\delta$ circuit is structured as an export over an export, reflecting that two arguments are expected. Notice the occurrences of these parameters in the 'function'-body of the export are arranged in function consumers. This is because the first part of the computation of average passes any supplied parameters to the $\lceil add \rceil_\varphi$ circuit, i.e. those parameters passed to $\lceil average \rceil_\delta$ will be supplied to $\lceil add \rceil_\varphi$. The result of the computed addition (which is passed to γ), and is the head of another argument-list; this list consumes the $\lceil divideTwo \rceil_\nu$ 'function' and sends its result, via some chaining of inputs and outputs, to δ.

We would like to highlight some differences between the $\mathcal{X}$-calculus reductions, and those reductions in $\overline{\lambda}\mu\tilde{\mu}$. We will therefore adopt a CBV *reduction strategy as done so in the $\overline{\lambda}\mu\tilde{\mu}$ reduction of the same interpreted program (see Example 2.3.23).*

The $\mathcal{X}$-term corresponding to the program $\lceil (average\ \mathbf{2})\mathbf{4} \rceil_\alpha$ is huge. We do not require the reader to parse or understand the entire term (shown below) but will highlight the key features of the term.

$$(\widehat{\boldsymbol{x}}(\widehat{\boldsymbol{y}}(\lceil divideTwo \rceil_\nu \widehat{\nu} \dagger \widehat{c}((\lceil add \rceil_\varphi \widehat{\varphi} \dagger \widehat{i}(\langle \boldsymbol{y} \cdot \circ \rangle \widehat{\circ}\,[i]\,\widehat{t}(\langle \boldsymbol{x} \cdot \lambda \rangle \widehat{\lambda}\,[t]\,\widehat{s}\langle s \cdot \gamma \rangle))))\widehat{\gamma}\,[c]\,\widehat{p}\langle p \cdot \sigma \rangle))\widehat{\sigma} \cdot \tau)\widehat{\tau} \cdot \delta)$$

$$\ldots\ \widehat{\delta} \dagger \widehat{k}\ \ldots$$

$$(\langle \mathbf{2} \cdot \mu \rangle \widehat{\mu}\,[k]\,\widehat{h}(\langle \mathbf{4} \cdot \theta \rangle \widehat{\theta}\,[h]\,\widehat{d}\langle d \cdot \alpha \rangle))$$

The term is split into three 'pieces'. The first part is the interpretation of the $\lceil average \rceil_\delta$ function that sends its output onto δ (as explained above). The third part is the function consumer expecting the first function on k; the term corresponds to an argument list containing the arguments in order $\mathbf{2}$ then $\mathbf{4}$; these are to be supplied to the $\lceil average \rceil_\delta$ circuit. In between, the components of the cut are shown, which expresses that the computation will involve connecting the $\lceil average \rceil_\delta$ export to the function consumer.

Using the reduction rules of the CBV *$\mathcal{X}$-calculus, the above term 'normalises' in 49 steps. Of course the majority of these steps are applications of the propagation rules. We*

will focus on the highlights of the reduction below, followed by an explanation.

$$
\begin{array}{lll}
 & & \lceil (average\ 2)4 \rceil_{\alpha} \\
 & = & \lceil average \rceil_{\delta} \widehat{\delta} \dagger \widehat{\kappa} (\langle \mathbf{2} \cdot \mu \rangle \widehat{\mu}\, [k]\, \widehat{h} (\langle \mathbf{4} \cdot \theta \rangle \widehat{\theta}\, [h]\, \widehat{d} \langle d \cdot \alpha \rangle)) \\
^{1} & \rightarrow^{*} & \lceil divideTwo \rceil_{\nu} \widehat{\nu} \dagger \widehat{c} ((\lceil add \rceil_{\varphi} \widehat{\varphi} \dagger \widehat{i} (\langle 4 \cdot \circ \rangle \widehat{\circ}\, [i]\, \widehat{t} (\langle 2 \cdot \lambda \rangle \widehat{\lambda}\, [t]\, \widehat{s} \langle s \cdot \gamma \rangle))) \widehat{\gamma}\, [c]\, \widehat{p} \langle p \cdot \alpha \rangle) \\
^{2} & \rightarrow^{*} & \lceil divideTwo \rceil_{\nu} \widehat{\nu} \dagger \widehat{c} ((\langle 4 \cdot \circ \rangle \widehat{\circ} \dagger \widehat{u} ((\widehat{v} \langle (u{+}v) \cdot \chi \rangle \widehat{\chi} \cdot v) \widehat{v} \dagger \widehat{t} (\langle 2 \cdot \lambda \rangle \widehat{\lambda}\, [t]\, \widehat{s} \langle s \cdot \gamma \rangle))) \widehat{\gamma}\, [c]\, \widehat{p} \langle p \cdot \alpha \rangle) \\
^{3} & \rightarrow^{*} & \lceil divideTwo \rceil_{\nu} \widehat{\nu} \dagger \widehat{c} (((\widehat{v} \langle (4{+}v) \cdot \chi \rangle \widehat{\chi} \cdot v) \widehat{v} \dagger \widehat{t} (\langle 2 \cdot \lambda \rangle \widehat{\lambda}\, [t]\, \widehat{s} \langle s \cdot \gamma \rangle)) \widehat{\gamma}\, [c]\, \widehat{p} \langle p \cdot \alpha \rangle) \\
^{4} & \rightarrow^{*} & \lceil divideTwo \rceil_{\nu} \widehat{\nu} \dagger \widehat{c} ((\langle 2 \cdot \lambda \rangle \widehat{\lambda} \dagger \widehat{v} (\langle (4{+}v) \cdot \chi \rangle \widehat{\chi} \dagger \widehat{s} \langle s \cdot \gamma \rangle)) \widehat{\gamma}\, [c]\, \widehat{p} \langle p \cdot \alpha \rangle) \\
^{5} & \rightarrow^{*} & \lceil divideTwo \rceil_{\nu} \widehat{\nu} \dagger \widehat{c} ((\langle 2 \cdot \lambda \rangle \widehat{\lambda} \dagger \widehat{v} \langle (4{+}v) \cdot \gamma \rangle) \widehat{\gamma}\, [c]\, \widehat{p} \langle p \cdot \alpha \rangle) \\
^{6} & \rightarrow & \lceil divideTwo \rceil_{\nu} \widehat{\nu} \dagger \widehat{c} (\langle (4{+}2) \cdot \gamma \rangle \widehat{\gamma}\, [c]\, \widehat{p} \langle p \cdot \alpha \rangle) \\
^{7} & = & \lceil divideTwo \rceil_{\nu} \widehat{\nu} \dagger \widehat{c} (\langle 6 \cdot \gamma \rangle \widehat{\gamma}\, [c]\, \widehat{p} \langle p \cdot \alpha \rangle) \\
^{8} & = & (\widehat{a} \langle (a/2) \cdot \pi \rangle \widehat{\pi} \cdot \nu) \widehat{\nu} \dagger \widehat{c} (\langle 6 \cdot \gamma \rangle \widehat{\gamma}\, [c]\, \widehat{p} \langle p \cdot \alpha \rangle) \\
^{9} & \rightarrow^{*} & \langle 6 \cdot \gamma \rangle \widehat{\gamma} \dagger \widehat{a} (\langle (a/2) \cdot \pi \rangle \widehat{\pi} \dagger \widehat{p} \langle p \cdot \alpha \rangle) \\
^{10} & \rightarrow^{*} & \langle (6/2) \cdot \pi \rangle \widehat{\pi} \dagger \widehat{p} \langle p \cdot \alpha \rangle \\
^{11} & = & \langle 3 \cdot \pi \rangle \widehat{\pi} \dagger \widehat{p} \langle p \cdot \alpha \rangle \\
^{12} & \rightarrow & \langle 3 \cdot \alpha \rangle
\end{array}
$$

We give the following commentary for the above reductions:

1. *The first set of reductions pass the argument list containing the numbers* 2 *and* 4 *to the argument list to be supplied to* $\lceil add \rceil_{\varphi}$*. Notice the order of the parameters are reversed as in the original example.*
2. $\lceil add \rceil_{\varphi}$ *is expanded to* $\widehat{u} (\widehat{v} \langle (u{+}v) \cdot \chi \rangle \widehat{\chi} \cdot v) \widehat{v} \cdot \varphi$*, and the argument list consumes the body of the outer export.*
3. *The first argument,* 4*, is supplied as input to the outer export 'function' of* $\lceil add \rceil_{\varphi}$*.*
4. *The output* v *of the partially evaluated add function* ($\widehat{v} \langle (4{+}v) \cdot \chi \rangle \widehat{\chi} \cdot v$) *is redirected to* γ*.*
5. *The argument list consumes the partially evaluated function resulting from the previous step.*
6. *The second argument,* 2*, is supplied to the add function.*
7. *(The meta-addition operation is performed).*
8. ($\lceil divideTwo \rceil_{\nu}$ *is expanded).*
9. *The second argument list consumes* $\lceil divideTwo \rceil_{\nu}$*.*
10. *The body of* $\lceil divideTwo \rceil_{\nu}$ *is passed the head of the argument list (the value* 6 *that is output on* γ*).*
11. *(The meta-division operation is performed).*
12. *The original output of* 3 *(i.e.,* γ*) is redirected to* α*.*

The above reductions describe, at a very low level of detail, the mechanical steps involved in evaluating the function $\lceil (average\ 2)4 \rceil_{\alpha}$*.*

Unlike the $\overline{\lambda}\mu\tilde{\mu}$-calculus, no eta-rules are needed for this reduction to return the

expected result[4]. The additional eta-reduction rules required by Curien and Herbelin's $\overline{\lambda}\mu\tilde{\mu}$-calculus are a result of the artificial construct (the stoup) present in underlying logical framework, but originally introduced by Parigot to try an obtain a uniqueness property on normal forms. The computational terms that correspond to the stoup are used to (selectively) 'name' $\overline{\lambda}\mu\tilde{\mu}$ terms. However, since they introduce auxiliary structure to a term, this structure must also be removed at some point during the computation, hence the need for eta-rules.

The approach in $\mathcal{X}$ is quite different; instead of adding special constructors to name selected terms, *every* term is named. Two consequences of this are: (i) there are more proof permutations in the underlying logic and (ii) the syntax is more verbose. To address the first point, we have already seen that permutations of term structure can be interesting to study (Example 3.1.15). To address the second point, we remind ourselves that we are not seeking the most succinct model of computation; our goal is to investigate the natural computational content of Classical Logic (within a sequent calculus framework). While it is true that the example does not reflect the most efficient method for computing the average of two numbers, it does in our opinion, reflect the most natural computational behaviour that arises from Gentzen's Sequent Calculus for Classical Logic. We will see in Chapter 6, how the approach of naming 'everything' leads to an entirely mechanical process for deriving computational terms.

3.1.3 On Strong-Normalisation

We should point out that using the rules of the pure $\mathcal{X}$-calculus, not all typeable term are strongly normalisable. For example, to propagate cuts over cuts immediately leads to non-termination, since we can always choose the outermost cut as the one to contract. Although the notion of cut-elimination as proposed here has no rule that would allow this behaviour, it can be mimicked, which can lead to non-termination for typeable terms, as already observed by Urban [86].

Take $P\widehat{\alpha} \dagger \widehat{x}(\langle x \cdot \beta \rangle \widehat{\beta} \dagger \widehat{z}Q)$, and assume $x \notin fs(Q), \beta \notin fp(P)$, and P, Q both pure,

[4]That is not to say that eta-rules should not be investigated within the Sequent Calculus framework

then:

$$\begin{array}{ll} P\widehat{\alpha} \dagger \widehat{x}(\langle x \cdot \beta \rangle \widehat{\beta} \dagger \widehat{z} Q) & \rightarrow (act\text{-R}), (\nwarrow cut) \\ (P\widehat{\alpha} \nwarrow \widehat{x}\langle x \cdot \beta \rangle)\widehat{\beta} \dagger \widehat{z}(P\widehat{\alpha} \nwarrow \widehat{x} Q) & \rightarrow (d\nwarrow), (\nwarrow gc) \\ (P\widehat{\alpha} \dagger \widehat{x}\langle x \cdot \beta \rangle)\widehat{\beta} \dagger \widehat{z} Q & \rightarrow (act\text{-L}), (cut\nearrow) \\ (P\widehat{\beta} \nearrow \widehat{z} Q)\widehat{\alpha} \dagger \widehat{x}(\langle x \cdot \beta \rangle \widehat{\beta} \nearrow \widehat{z} Q) & \rightarrow (\nearrow d), (\nearrow gc) \\ P\widehat{\alpha} \dagger \widehat{x}(\langle x \cdot \beta \rangle \widehat{\beta} \dagger \widehat{z} Q) & \end{array}$$

(example communicated by Alexander J. Summers)

Cut-elimination involving capsules in this way represent a special case; capsules are the only term constructors that introduce two connectors. The calculus features cut-over-cut propagation (to simulate full β-reduction) and so in the case above, where two cuts "fight" to connect to the different connectors of the capsule, they will continue propagating over each other forever under an outermost reduction strategy.

Urban gives a solution for this unwanted reduction behaviour, and shows it sufficient to obtain strong-normalisation of typeable terms. He adds the rules:

$$\begin{array}{l} (P\widehat{\alpha} \dagger \widehat{x}\langle x \cdot \beta \rangle)\widehat{\beta} \nearrow \widehat{y} Q \rightarrow (P\widehat{\beta} \nearrow \widehat{y} Q)\widehat{\alpha} \dagger \widehat{y} Q \\ P\widehat{\alpha} \nwarrow \widehat{x}(\langle x \cdot \beta \rangle \widehat{\beta} \dagger \widehat{y} Q) \rightarrow P\widehat{\alpha} \dagger \widehat{y}(P\widehat{\alpha} \nwarrow \widehat{x} Q) \end{array}$$

and gives them priority over the rules $(cut\nearrow)$ and $(\nwarrow cut)$ by changing those to

$$\begin{array}{l} (P\widehat{\alpha} \dagger \widehat{x} Q)\widehat{\beta} \nearrow \widehat{y} R \rightarrow (P\widehat{\beta} \nearrow \widehat{y} R)\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\beta} \nearrow \widehat{u} R) \leftarrow Q \neq \langle x \cdot \beta \rangle \\ P\widehat{\alpha} \nwarrow \widehat{x}(Q\widehat{\beta} \dagger \widehat{y} R) \rightarrow (P\widehat{\alpha} \nwarrow \widehat{x} Q)\widehat{\beta} \dagger \widehat{y}(P\widehat{\alpha} \nwarrow \widehat{x} R) \leftarrow Q \neq \langle x \cdot \beta \rangle \end{array}$$

However, notice that the side-condition $Q \neq \langle x \cdot \beta \rangle$ is quite different in character from the rules for $\mathcal{X}$ we presented above, in that now *equality* between circuits is tested, rather than just a syntactic property of a circuit.

In our implementation, we have chosen a slightly different approach: we avoid deactivation of cuts. This implies that we remove the rules $(\nearrow d)$ and $(d\nwarrow)$, and add the following rules (notice that we do not need to check if a circuit matches another, nor need to give priority to rules):

$$\begin{array}{lll} (flip\nwarrow): & \langle z \cdot \alpha \rangle \widehat{\alpha} \nearrow \widehat{x} P \rightarrow \langle z \cdot \alpha \rangle \widehat{\alpha} \nwarrow \widehat{x} P \leftarrow & P \text{ does not introduce } x \\ (\nearrow imp): & \langle y \cdot \alpha \rangle \widehat{\alpha} \nearrow \widehat{x}(P\widehat{\delta}[x]\,\widehat{z} Q) \rightarrow P\widehat{\delta}[y]\,\widehat{z} Q \leftarrow & x \text{ introduced} \\ (\nearrow cap): & \langle z \cdot \alpha \rangle \widehat{\alpha} \nearrow \widehat{x}\langle x \cdot \beta \rangle \rightarrow \langle z \cdot \beta \rangle & \end{array}$$

$$
\begin{array}{lll}
(\nleftarrow flip): & P\widehat{\alpha} \nwarrow \widehat{x}\langle x\cdot\beta\rangle \to P\widehat{\alpha} \nleftarrow \widehat{x}\langle x\cdot\beta\rangle & \leftarrow \textit{ P does not introduce } \alpha \\
(exp\nwarrow): & (\widehat{y}P\widehat{\delta}\cdot\alpha)\widehat{\alpha} \nwarrow \widehat{x}\langle x\cdot\gamma\rangle \to \widehat{y}P\widehat{\delta}\cdot\gamma & \leftarrow \textit{ } \alpha \textit{ introduced} \\
(cap\nwarrow): & \langle x\cdot\alpha\rangle\widehat{\alpha} \nwarrow \widehat{y}\langle y\cdot\beta\rangle \to \langle x\cdot\beta\rangle &
\end{array}
$$

These rules introduce an additional feature we will exploit in Chapter 4: they allow renamings to be prioritized over other reductions.

3.1.4 Optimising Reduction

The set of rules of the pure $\mathcal{X}$-calculus can be optimised further. For example, the applicability of the garbage collection rules stated in Lemma 3.1.8 is limited, since they both involve pure terms. Here we are generalise these results to include terms with active cuts.

We aim to add more generic rules; in fact, we will show their admissibility below (Theorem 3.1.21), for which we first need to show a number of results.

Lemma 3.1.17 (Elimination of Active Cuts) *Every circuit whose root is an active cut can be reduced to a pure circuit. i.e.,*

1. *For all P, Q pure, there exists an R pure such that $P\widehat{\alpha} \nleftarrow \widehat{x}Q \to R$.*
2. *For all P, Q pure, there exists an R pure such that $P\widehat{\alpha} \nwarrow \widehat{x}Q \to R$.*

Proof 3.1.18 *By induction on structure of circuits.*

1. *We highlight one case, where P, Q are pure:*

$$
\begin{array}{lll}
(P = \widehat{y}P'\widehat{\beta}\cdot\alpha): & (\widehat{y}P'\widehat{\beta}\cdot\alpha)\widehat{\alpha} \nleftarrow \widehat{x}Q & \to (\textit{exp-outs}\nleftarrow) \\
& (\widehat{y}(P'\widehat{\alpha} \nleftarrow \widehat{x}Q)\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}Q & \to (IH) \\
& (\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}Q &
\end{array}
$$

   Notice that this last term is pure. □

2. *We highlight again one case, where P, Q_1, Q_2 are pure:*

$$
\begin{array}{lll}
Q = Q_1\widehat{\beta}\,[x]\,\widehat{y}Q_2: & P\widehat{\alpha} \nwarrow \widehat{x}(Q_1\widehat{\beta}\,[x]\,\widehat{y}Q_2) & \to (\nwarrow \textit{imp-outs}) \\
& P\widehat{\alpha} \dagger \widehat{v}((P\widehat{\alpha} \nwarrow \widehat{x}Q_1)\widehat{\beta}\,[v]\,\widehat{y}(P\widehat{\alpha} \nwarrow \widehat{x}Q_2)) & \to (IH) \\
& P\widehat{\alpha} \dagger \widehat{v}(R_1\widehat{\beta}\,[v]\,\widehat{y}R_2) &
\end{array}
$$

   Notice again that this last term is pure. □

We can now use this lemma to give a stronger result.

Lemma 3.1.19 (Every circuit is reducible to a pure circuit) *For all $\mathcal{X}$-circuits P, there exists a reduction path $P \to P'$, with P' pure.*

Proof 3.1.20 *For each active cut in a circuit, we define its* depth, *d, as the distance (calculated in nodes) from the root of the tree. For any particular depth of the circuit, we define its* group size, *g, as the number of active cuts at that depth of the tree. We define the* class, *c, of a circuit as the pair of the depth of the innermost cut and its group size: $c = \langle d, g \rangle$.*

We finish the proof by lexicographic induction on the class of a circuit.

If P is pure, then $P \equiv P'$. Otherwise: take the set (of size g) of innermost active cuts at depth d. There are two cases to consider:

- *if $g > 1$, take any circuit in this set $R\widehat{\alpha} \nleftarrow \widehat{x}Q$ (or $R\widehat{\alpha} \nwarrow \widehat{x}Q$). R and Q are known to be pure; otherwise they would not be the innermost flagged cuts. By Lemma 3.1.17, we know the circuit reduces to some other circuit S, which is pure. This eliminates the active cut from the proof, so the group-size of the circuit reduces by one, and the class of the circuit decreases.*
- *if $g = 1$, the active cut is eliminated from the circuit by Lemma 3.1.17. Since there are no more active cuts at this level, the depth of the innermost cut decreases (to the next lowest), and the class of the circuit reduces.* □

We are now ready to justify some more general garbage collection rules, that essentially equates the nets $P\widehat{\alpha} \nleftarrow \widehat{x}Q$ and P, provided $\alpha \notin fp(P)$.

Lemma 3.1.21 (Generalised Garbage Collection) *We will show the following two properties, which generalise the garbage collection rules given in Lemma 3.1.8 to non-pure circuits.*

$$(\nleftarrow gc): \quad P\widehat{\alpha} \nleftarrow \widehat{x}Q \overset{\mathcal{X}}{=} P \;\leftarrow\; \alpha \notin fp(P) \qquad (1)$$

$$(\nwarrow gc): \quad P\widehat{\alpha} \nwarrow \widehat{x}Q \overset{\mathcal{X}}{=} Q \;\leftarrow\; x \notin fs(Q) \qquad (2)$$

Recall that $P \overset{\mathcal{X}}{=} Q$ iff the circuits P and Q have the same set of normal forms.

Proof 3.1.22 *1. For the second part, that each normal form T of P is a normal form of $P\widehat{\alpha} \nleftarrow \widehat{x}Q$, we reduce $P\widehat{\alpha} \nleftarrow \widehat{x}Q$ to $T\widehat{\alpha} \nleftarrow \widehat{x}Q$, remark that T is pure, and apply rule $(\nleftarrow gc)$.*

For the first, we show that if $P\widehat{\alpha} \not\leftarrow \widehat{x}Q \to T$, then $P \to T$, where T is a normal form. We achieve this by showing that we can run a reduction on P, mimicking a reduction taking place on $P\widehat{\alpha} \not\leftarrow \widehat{x}Q$ by essentially ignoring all reductions inside Q; the only problem might be when the presence of $[\,]\widehat{\alpha} \not\leftarrow \widehat{x}Q$ disturbs the reduction behaviour.

The proof completes by co-induction (we only show some interesting cases):

• *$P = (P_1\widehat{\beta}\,[v]\,\widehat{y}P_2)\widehat{\gamma}\dagger\widehat{z}P_3$. We can run $((P_1\widehat{\beta}\,[v]\,\widehat{y}P_2)\widehat{\gamma}\dagger\widehat{z}P_3)\widehat{\alpha} \not\leftarrow \widehat{x}Q$ in a number of ways. Any reduction inside P_1, P_2 or P_3 is dealt with by induction, so we can focus on the cuts involved. Assume we apply rule $(cut\not\leftarrow)$ to propagate the outermost cut and obtain*

$$((P_1\widehat{\beta}\,[v]\,\widehat{y}P_2)\widehat{\alpha} \not\leftarrow \widehat{x}Q)\widehat{\gamma}\dagger\widehat{z}(P_3\widehat{\alpha} \not\leftarrow \widehat{x}Q)$$

Now the top-most (inactive) cut can be activated in two directions; let's assume we decided to go left:

$$((P_1\widehat{\beta}\,[v]\,\widehat{y}P_2)\widehat{\alpha} \not\leftarrow \widehat{x}Q)\widehat{\gamma} \not\leftarrow \widehat{z}(P_3\widehat{\alpha} \not\leftarrow \widehat{x}Q)$$

This activated cut cannot propagate, since the cut directly underneath it is active; propagating that first gives

$$((P_1\widehat{\alpha} \not\leftarrow \widehat{x}Q)\widehat{\beta}\,[v]\,\widehat{y}(P_2\widehat{\alpha} \not\leftarrow \widehat{x}Q))\widehat{\gamma} \not\leftarrow \widehat{z}(P_3\widehat{\alpha} \not\leftarrow \widehat{x}Q)$$

Now the top-cut can propagate, to give

$$((P_1\widehat{\alpha} \not\leftarrow \widehat{x}Q)\widehat{\gamma} \not\leftarrow \widehat{z}(P_3\widehat{\alpha} \not\leftarrow \widehat{x}Q))\widehat{\beta}\,[v]\,\widehat{y}((P_2\widehat{\alpha} \not\leftarrow \widehat{x}Q)\widehat{\gamma} \not\leftarrow \widehat{z}(P_3\widehat{\alpha} \not\leftarrow \widehat{x}Q))$$

By induction we can mimic $P_i\widehat{\alpha} \not\leftarrow \widehat{x}Q$ by P_i, for $i \in \{1,2,3\}$. We can simulate this particular reduction on P as follows:

$$(P_1\widehat{\beta}\,[v]\,\widehat{y}P_2)\widehat{\gamma}\dagger\widehat{z}P_3 \;\to (act\text{-}\mathrm{L}), (imp\not\leftarrow)\; (P_1\widehat{\gamma} \not\leftarrow \widehat{z}P_3)\widehat{\beta}\,[v]\,\widehat{y}(P_2\widehat{\gamma} \not\leftarrow \widehat{z}P_3)$$

• *$P = (\widehat{y}P_1\widehat{\beta}\cdot\gamma)\widehat{\gamma}\dagger\widehat{z}P_2$. As above, a reduction inside P_1 or P_2 creates no problems, so we can focus on the cuts involved. Assume we decide propagate the top cut, and get*

$$((\widehat{y}P_1\widehat{\beta}\cdot\gamma)\widehat{\alpha} \not\leftarrow \widehat{x}Q)\widehat{\gamma}\dagger\widehat{z}(P_2\widehat{\alpha} \not\leftarrow \widehat{x}Q)$$

If we now left-activate the top cut, similar to above, we can only propagate the

innermost cut, and obtain:

$$(\widehat{y}(P_1\widehat{\alpha} \nearrow \widehat{x}Q)\widehat{\beta}\cdot\gamma)\widehat{\gamma} \nearrow \widehat{z}(P_2\widehat{\alpha} \nearrow \widehat{x}Q)$$

Now applying rule (exp-outs$\nearrow$) will give:

$$(\widehat{y}((P_1\widehat{\alpha} \nearrow \widehat{x}Q)\widehat{\gamma} \nearrow \widehat{z}(P_2\widehat{\alpha} \nearrow \widehat{x}Q))\widehat{\beta}\cdot\delta)\widehat{\delta} \dagger \widehat{z}(P_2\widehat{\alpha} \nearrow \widehat{x}Q)$$

Now, if γ is introduced in $\widehat{y}P_1\widehat{\beta}\cdot\gamma$, then γ does not appear free inside P_1 nor in Q, and by induction we can assume both that the term $(P_1\widehat{\alpha} \nearrow \widehat{x}Q)\widehat{\gamma} \nearrow \widehat{z}(P_2\widehat{\alpha} \nearrow \widehat{x}Q)$ can be simulated by P_1, and that $P_2\widehat{\alpha} \nearrow \widehat{x}Q$ can be simulated by P_2. We can mimic this reduction with a reduction path of length zero, performing an α-conversion:

$$(\widehat{y}P_1\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}P_2 \to (\widehat{y}P_1\widehat{\beta}\cdot\delta)\widehat{\delta} \dagger \widehat{z}P_2$$

If γ is not introduced, we still can assume that $P_i\widehat{\alpha} \nearrow \widehat{x}Q$ can be simulated by P_i, for $i \in \{1, 2\}$, and we can simulate this particular reduction on P via a series of steps:

$$(\widehat{y}P_1\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}P_2 \to (\widehat{y}P_1\widehat{\beta}\cdot\gamma)\widehat{\gamma} \nearrow \widehat{z}P_2 \to (\widehat{y}(P_1\widehat{\gamma} \nearrow \widehat{z}P_2)\widehat{\beta}\cdot\delta)\widehat{\delta} \dagger \widehat{z}P_2$$

- *All other cases are shown in a similar fashion.*

So every reduction to a normal form, starting from $P\widehat{\alpha} \nearrow \widehat{x}Q$, can be mimicked by reducing P, so every normal form, reachable from $P\widehat{\alpha} \nearrow \widehat{x}Q$, can be reached from P.

2. *Similar.* □

This result now helps to justify more general deactivation rules.

Theorem 3.1.23 (Generalised Deactivation) *We will show the following properties.*

1. $P\widehat{\alpha} \nearrow \widehat{x}Q \overset{\mathcal{X}}{=} P\widehat{\alpha} \dagger \widehat{x}Q \leftarrow$ *α introduced*
2. $P\widehat{\alpha} \nwarrow \widehat{x}Q \overset{\mathcal{X}}{=} P\widehat{\alpha} \dagger \widehat{x}Q \leftarrow$ *x introduced*

Proof 3.1.24 *1. If P introduces α, we have two cases:*

$P = \langle x\cdot\alpha\rangle$: By rule ($\nearrow d$).

$P = \widehat{x}P'\widehat{\beta}\cdot\alpha$: Then $\alpha \notin fp(P)$, and

$$\begin{array}{ll} & (\widehat{x}P'\widehat{\beta}\cdot\alpha)\widehat{\alpha} \nearrow \widehat{x}Q \\ \to & (exp\text{-}outs\nearrow) \\ & (\widehat{x}(P'\widehat{\alpha} \nearrow \widehat{x}Q)\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{x}Q \\ \overset{\mathcal{X}}{=} & (Lemma\ 3.1.21) \\ & (\widehat{x}P'\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{x}Q \\ =_\alpha & (\widehat{x}P'\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}Q \end{array}$$

2. *If Q introduces x, we have two cases:*

   $Q = \langle x{\cdot}\beta \rangle$: *By rule* $(d\nwarrow)$.

   $Q = Q_1 \widehat{\beta}\,[x]\,\widehat{y} Q_2$: *Then* $x \notin fs(Q_1, Q_2)$, *and*

$$
\begin{array}{ll}
P'\widehat{\alpha} \nwarrow \widehat{x}(Q_1 \widehat{\beta}\,[x]\,\widehat{y} Q_2) & \to (\nwarrow \textit{imp-outs}) \\
P'\widehat{\alpha} \dagger \widehat{v}((P'\widehat{\alpha} \nwarrow \widehat{x} Q_1)\widehat{\beta}\,[v]\,\widehat{y}(P'\widehat{\alpha} \nwarrow \widehat{x} Q_2)) & \overset{\mathcal{X}}{=} (\textit{Lemma 3.1.21}) \\
P'\widehat{\alpha} \dagger \widehat{v}(Q_1 \widehat{\beta}\,[v]\,\widehat{y} Q_2) & =_{\alpha} \; P'\widehat{\alpha} \dagger \widehat{x}(Q_1 \widehat{\beta}\,[x]\,\widehat{y} Q_2)
\end{array}
$$

□

Similarly, we can also show:

Lemma 3.1.25 (Generalised Renaming) *We can show the following properties, which generalise the renaming rules given in Lemma 3.1.8 to non-pure circuits.*

$$
\begin{array}{ll}
(\textit{ren}\text{-L}): & P\widehat{\alpha} \nearrow \widehat{x}\langle x{\cdot}\beta \rangle \overset{\mathcal{X}}{=} P[\beta/\alpha] \\
(\textit{ren}\text{-R}): & \langle y{\cdot}\alpha \rangle \widehat{\alpha} \nwarrow \widehat{x} P \overset{\mathcal{X}}{=} P[y/x]
\end{array}
$$

In Section 4.3, we will review the impact on reduction cost when adding these more generic reduction rules.

3.2 Chapter Summary

In this chapter, we introduced the $\mathcal{X}$-calculus of van Bakel, Lengrand and Lescanne [9]. We studied its most prominent features, namely its novel symmetric syntax and its reduction mechanism (which does not follow the usual notion of term-for-variable substitution). We related these features to well-understood computational notions, and gave some comparisons with the λ-calculus and with Curien and Herbelin's $\overline{\lambda}\mu\tilde{\mu}$-calculus.

The $\mathcal{X}$-calculus is non-confluent. We showed, through examples, that while this is traditionally seen as an undesired property for computational term-calculi (especially when considering functional programming) its presence in the $\mathcal{X}$-calculus leads to subject matter worth investigating. In particular, two confluent reduction subsystems can be defined within the pure $\mathcal{X}$-calculus, namely: the call-by-name and call-by-value subsystems.

We proposed a modification to the system to regain strong-normalisation of typed terms. Finally, we optimised some reduction behaviours of the $\mathcal{X}$-calculus by giving some general notions of garbage collection, renaming and 'deactivation'.

Chapter 4

Implementing $\mathcal{X}$

In the background chapter (Section 2.4), we looked at some sound approaches (w.r.t. reduction) of implementing higher-order rewrite systems. As discussed, when dealing with systems which have variable bindings, care must be taken during the transformation of programs to ensure the *variable binding* and *variable identity* relationships are preserved[1]. The most commonly studied system in which these relationships are present is the λ-calculus. Computation in this calculus (specified by the β-reduction rule) has the overall effect of performing a (capture-free) 'term for variable' substitution operation. Many higher-order implementations internalise this operation, and deal with the problem of variable-capture *behind the scenes*. This is true of all of the systems studied, with the exception of explicit substitutions.

In Chapter 3, we studied the overall effect of computation in the $\mathcal{X}$-calculus and observed that it did *not* correspond to the usual notion of 'term for variable' substitution. This fact eliminated the possibility of directly adopting many existing implementation techniques, like de Bruijn indices and Wolfram's second order term graphs. Of course, each of the approaches we reviewed can be extended (or modified by some means) to implement $\mathcal{X}$, but now this leaves a choice of which system to extend.

Notice that the $\mathcal{X}$-calculus is a rewrite system with side conditions on the rewrite rules. Wadsworth (Section 2.4.4) describes a simple approach to computing free-variable checks using sets and a notion of *paths* on his λ-graphs. The $\mathcal{X}$-calculus requires the additional specification of the *introduces* side-condition. Whichever formalism we choose to implement the $\mathcal{X}$-calculus, it will need to be extended to

[1] Recall that the variable binding relation associates a formal parameter with its occurrences in the body of the subterm, while the variable identity relation equates free variables.

express at least these two side-conditions.

We prefer not to work with de Bruijn indices, for many of the disadvantages listed in Section 2.4.3. De Bruijn indices were traditionally invented to implement the capture-free 'term for variable' substitution, though we are also aware of the more general system of higher-order rewriting using de Bruijn notation of Bonelli *et al.* [19]. The $\mathcal{X}$-calculus is already complicated to unfamiliar eyes, and a de Bruijn notation would certainly further add to this complexity. In addition, the $\mathcal{X}$-calculus is (at the time of writing) a fairly young calculus. One of the goals of the implementation was to better understand its features, in particular the reduction mechanism; we reported on some of our insights in the previous chapter.

Kahl's second-order graphs provide a clean implementation for higher-order rewrite systems, again, internalising a capture-free 'term for variable' notion of substitution. This is true of all implementations of Klop's CRS that we are aware of. A particularly nice feature of Kahl's system is the explicit representation of the variable binding and variable identity relations. While this system is perhaps the best suited to the implementation of $\mathcal{X}$, we avoid it for practical reasons: the implementation appears to be closed source, and additionally we will not require the full power of Klop's CRS's formalism.

The approach we take instead is to extend the traditional first order term-graph rewrite systems due to Barendregt *et al.* (presented in Section 2.4.5) with binding constructs and side-conditions. This system has been well-studied, and is (relatively) simple to understand. Moreover, full implementation details are very easy to get hold of. In this chapter section, we will describe specifically the extensions we needed to model $\mathcal{X}$-calculi, and detail key design decisions we made in our implementation.

4.1 Conditional Second-Order Term Graph Rewriting

A conditional second-order term graph rewrite system (CTGRS) is an extension of the traditional first-order term graph rewrite systems. The extension allows for: (i) a representation of the variable binding and variable identity relations and, (ii) the ability to express and check side-conditions.

Definition 4.1.1 (CTGRS) *A conditional term graph rewrite system is a pair* $\langle \Sigma, \mathcal{R} \rangle$*, where* Σ *is a signature and* $\mathcal{R}$ *is a set of graph rewrite rules.*

We first define the alphabet over which term graphs will be built; this is an extension of the usual signature (see Definition 2.4.16) with two new concepts. We introduce a notion of *name* which corresponds to the formal parameters of a higher-order function (e.g., the 'variables' of the λ-calculus as used in the abstraction, or the connectors of the $\mathcal{X}$-calculus). We make a second extension of the signature, introducing a notion of *binder specification*s on function symbols; these are intended to relate a formal parameter its occurrences in the subterm it binds over (recall that in the $\mathcal{X}$-calculus, terms have several subterms and several binders).

Definition 4.1.2 (Signature) *The signature, Σ, of a* CTGRS *is a 5-tuple*

$$\langle \mathcal{F}, \mathcal{N}, \mathcal{V}, arity, \mathcal{B} \rangle$$

where:

- *$\mathcal{F}$ is a non-empty set of function symbols.*
- *$\mathcal{N}$ is a countably infinite set of* name *symbols.*
- *$\mathcal{V}$ is a countably infinite set of metavariables.*
- *arity is a function of type $\mathcal{F} \rightarrow \mathbb{N}$, specifying the number of arguments each function symbol can take.*
- *$\mathcal{B}$ is a* binder specification *function of type $\{\mathcal{F} \times \mathbb{N} \times \mathbb{N}\}$, relating (the indexes of) two different arguments of the specified function symbol. (The intended use is given some $\langle fs, i, j \rangle \in \mathcal{B}$, the i^{th} argument of f is a binder over the subterm at index j).*

Note that the sets $\mathcal{F}, \mathcal{N}$ and $\mathcal{V}$ are disjoint.

Definition 4.1.3 (Term Graph) *A rooted labelled graph over the signature Σ is a 6-tuple*

$$\langle X, lab, F, N, succ, z \rangle$$

where:

- *$X \in \mathbb{N}$ is a set of nodes.*
- *lab is a function of type $X \rightarrow (\Sigma_{\mathcal{F}} \cup \Sigma_{\mathcal{N}})$, mapping each node in X to a function symbol or name symbol.*
- *F, N are disjoint partitions that cover the set of graph nodes X, where:*

$$F = \left\{ n \in X \mid \Sigma_{lab(n)} \in \mathcal{F} \right\}$$

$$N = \left\{ n \in X \mid \Sigma_{lab(n)} \in \mathcal{N} \right\}$$

- *succ is a function of type $X \rightarrow [X]$, specifying an ordered list of arguments for each node in X. The i^{th} argument of a node $n \in X$ with arity k is denoted $succ(n)_i$, where $0 \leq i < k$.*
- *$z \in F$ is the unique root of the graph.*

We do not require that every node is reachable from the root of the graph.

Definition 4.1.4 (Open Graph) *An* open graph *is the 6-tuple $\langle X, lab, F, N, V, succ \rangle$ like a term graph, except:*

- *no root is specified.*
- *$V = \left\{ n \in X \mid \Sigma_{lab(n)} \in \mathcal{V} \right\}$ and is pairwise disjoint with F and N.*
- *succ is required to be only a partial function, where $\forall v \in V . succ(v)$ is undefined.*

Elements of V are 0-ary metavariables or open nodes *that will be mapped to elements of F. When we write open graphs, we will specify a set of infinite symbols over which open nodes range. We say a graph is closed when it contains no open nodes.*

In the following, we will afford ourselves the liberty of treating term graphs as rooted *open graphs according to the following conversion.*

$$\langle X, lab, F, N, succ, z \rangle = \langle X, lab, F, N, \emptyset, succ, z \rangle$$

We use the standard definitions of *paths* and *subgraphs* as given in Section 2.4.5, except that they are defined (where applicable) over our new signature and graph structures.

Graph rewrite rules may additionally contain side-conditions. We remind ourselves that these side-conditions apply to *instances of* left-hand sides of rewrite rules, rather than on the rewrite rules themselves. With this in mind we define a *specification* for a side condition.

Definition 4.1.5 (Side-Condition Specification) *A side-condition specification is defined with respect to an open graph, $g = \langle X, lab, F, N, V, succ \rangle$. We will use the variables x, y to range over X and the variables m to range over N. We define the set of*

side-condition specifications with the following grammar.

$$
\begin{array}{rll}
SCS ::= & \mathit{equals}(x,y) & \text{`}x = y\text{'} \\
| & \mathit{fv}(m,x) & \text{`}m \in \mathit{fv}(x)\text{'} \\
| & \mathit{bv}(m,x) & \text{`}m \in \mathit{bv}(x)\text{'} \\
| & \mathit{introduces}(x,m) & \text{`}x \text{ introduces } m\text{'} \\
| & \mathit{or}(SCS,SCS) & \text{`disjunction'} \\
| & \mathit{and}(SCS,SCS) & \text{`conjunction'} \\
| & \mathit{not}(SCS) & \text{`negation'} \\
| & \mathit{true} & \text{`true'} \\
| & \mathit{false} & \text{`false'}
\end{array}
$$

We will specify how to evaluate instances of these specifications in Definition 4.1.7.

Definition 4.1.6 (Graph Rewrite Rule) *The graph representation of a conditional rewrite rule is a bi-rooted open graph called a* graph rewrite rule*, and is denoted by the quadruple $\langle g, l, r, sc \rangle$, where g is an open graph (represented as a 6-tuple), and l and r are nodes in g_F called the* left root *and* right root *of the rule. A side-condition, sc, is associated with each rule. If no side-condition is specified, the default side-condition true is used.*

We will also reuse the standard definitions of *homomorphism*s and *term graph substitution*s from Section 2.4.5, adapted to our needs.

We may now specify how to evaluate a side-condition specification with respect to a term graph and a homomorphism from the left-hand side of a graph rewrite rule to that term graph.

Definition 4.1.7 (Evaluation of Side-Condition Specifications) *We can define a procedure* `eval`*, which evaluates the side-condition specification of the rewrite rule with respect to the following structures:*

- *a graph rewrite rule, $\langle R, l, r, sc \rangle$.*
- *a term graph, T.*
- *a homomorphism, $\partial : R_X \rightarrow T_X$ from the nodes in R_X reachable from the left root, to a subset of the nodes in T_X.*

We will first define the following auxiliary functions, with respect to the signature Σ of graphs:

- `subTerms` *returns the subterm node indexes of the supplied function symbol node.*

$$\texttt{subTerms} :: F \to \mathit{Open\ Graph} \to \{\mathbb{N}\}$$
$$\texttt{subTerms}\ n\ g = \left\{ i \mid g_{succ_i(n)} \in g_F \wedge 0 \leq i < \Sigma_{arity(n)} \right\}$$

- `names` *returns the name node indexes of the supplied function symbol node.*

$$\texttt{names} :: F \to \mathit{Open\ Graph} \to \{\mathbb{N}\}$$
$$\texttt{names}\ n\ g = \left\{ i \mid g_{succ_i(n)} \in g_N \wedge 0 \leq i < \Sigma_{arity(n)} \right\}$$

- `binders` *returns the node id's of binders over the supplied subterm.*

$$\texttt{binders} :: F \to \mathbb{N} \to \mathit{Open\ Graph} \to \{X\}$$
$$\texttt{binders}\ f\ g\ s = \left\{ g_{lab(i)} \mid \langle f, i, s \rangle \in \Sigma_{\mathcal{B}} \right\}$$

- `fnSet` *computes the set of* free *names in the supplied subgraph* $g|n$.

$$\begin{aligned}
&\texttt{fnSet} :: F \to \mathit{Open\ Graph} \to \{N\} \\
&\texttt{fnSet}\ n\ g = \mathit{vars} \cup \mathit{fnSubterms} \\
&\quad \texttt{where} \\
&\qquad \mathit{vars} = \left\{ g_{succ_i(n)} \mid i \in \texttt{names}\ n\ g \wedge \neg \exists j. \langle g_{lab(n)}, i, j \rangle \in \Sigma_{\mathcal{B}} \right\} \\
&\qquad \mathit{fnSubterms} = \{ x \in (\texttt{fnSet}\ s\ g \setminus \texttt{binders}\ n\ s\ g) \mid s \in \texttt{subterms}\ n\ g \}
\end{aligned}$$

- `bnSet` *computes the set of* bound *names in the supplied subgraph* $g|n$.

$$\begin{aligned}
&\texttt{bnSet} :: F \to \mathit{Open\ Graph} \to \{N\} \\
&\texttt{bnSet}\ n\ g = \mathit{vars} \cup \mathit{bnSubterms} \\
&\quad \texttt{where} \\
&\qquad \mathit{vars} = \left\{ g_{succ_i(n)} \mid i \in \texttt{names}\ n\ g \wedge \exists j. \langle g_{lab(n)}, i, j \rangle \in \Sigma_{\mathcal{B}} \right\} \\
&\qquad \mathit{bnSubterms} = \left\{ x \in \texttt{bnSet}\ g_{succ_j(n)}\ g \mid j \in \texttt{subterms}\ n\ g \right\}
\end{aligned}$$

- `introSet` *computes the set of* introduced *names in the supplied subgraph* $g|n$.

$$\begin{aligned}
&\texttt{introSet} :: F \to \mathit{Open\ Graph} \to \{N\} \\
&\texttt{introSet}\ n\ g = \mathit{vars} \setminus \mathit{fnSubterms} \\
&\quad \texttt{where} \\
&\qquad \mathit{vars} = \left\{ g.succ_i(n) \mid i \in \texttt{names}\ n\ g \wedge \neg \exists j. \langle g_{lab(n)}, i, j \rangle \in \Sigma_{\mathcal{B}} \right\} \\
&\qquad \mathit{fnSubterms} = \{ x \in (\texttt{fnSet}\ s\ g \setminus \texttt{binders}\ n\ s\ g) \mid s \in \texttt{subterms}\ n\ g \}
\end{aligned}$$

Where the variables x, y *range over* R_X, m *ranges over* R_N, *and* c_1, c_2 *range over SCS,*

we define the `eval` *function as,*

$$
\begin{array}{ll}
\texttt{eval} :: \mathit{SCS} \rightarrow \mathit{Open\ Graph} \rightarrow (R_X \rightarrow T_X) \rightarrow \texttt{boolean} & \\
\texttt{eval}\ \mathit{equals}(x,y)\ g\ \partial & = (\partial(x) == \partial(y)) \\
\texttt{eval}\ \mathit{fv}(m,x)\ g\ \partial & = \partial(m) \in \texttt{fnSet}\ \partial(x)\ g \\
\texttt{eval}\ \mathit{bv}(m,x)\ g\ \partial & = \partial(m) \in \texttt{bnSet}\ \partial(x)\ g \\
\texttt{eval}\ \mathit{introduces}(x,m)\ g\ \partial & = \partial(m) \in \texttt{introSet}\ \partial(x)\ g \\
\texttt{eval}\ \mathit{and}(c_1,c_2)\ g\ \partial & = (\texttt{eval}\ c_1\ g\ \partial) \wedge (\texttt{eval}\ c_2\ g\ \partial) \\
\texttt{eval}\ \mathit{or}(c_1,c_2)\ g\ \partial & = (\texttt{eval}\ c_1\ g\ \partial) \vee (\texttt{eval}\ c_2\ g\ \partial) \\
\texttt{eval}\ \mathit{not}(c_1)\ g\ \partial & = \neg(\texttt{eval}\ c_1\ g\ \partial) \\
\texttt{eval}\ \mathit{true}\ g\ \partial & = \mathit{true} \\
\texttt{eval}\ \mathit{false}\ g\ \partial & = \mathit{false}
\end{array}
$$

The evaluation of side conditions have been formulated to yield correct results on terms with nested binding.

Definition 4.1.8 (Redex) *A* redex *in a term graph g_0 is a pair $\langle R, \partial \rangle$, where R is a graph rewrite rule $\langle g, l, r, sc \rangle$, ∂ is a homomorphism from $(g|l)$ to g_0, and* `eval` *$sc\ g_0\ \partial$ evaluates to true.*

The standard graph rewrite step defined in Section 2.4.5 is used to transform CTGRS graphs, i.e., only the definition of a redex has changed.

4.1.1 A CTGRS specification of the $\mathcal{X}$-calculus

In this section we will define the $\mathcal{X}$-calculus as an example of a CTGRS. Terms of the $\mathcal{X}$-calculus can, of course, be written using the more traditional prefix syntax as shown below.

$\mathcal{X}$-*syntax*	*Prefix notation*
$\langle x \cdot \alpha \rangle$	$Cap(x, \alpha)$
$\widehat{x} P \widehat{\beta} \cdot \alpha$	$Exp(x, P, \beta, \alpha)$
$P \widehat{\alpha}\ [x]\ \widehat{y} Q$	$Imp(P, \alpha, x, y, Q)$
$P \widehat{\alpha} \dagger \widehat{x} Q$	$Cut(P, \alpha, x, Q)$
$P \widehat{\alpha} \not\succ \widehat{x} Q$	$CutR(P, \alpha, x, Q)$
$P \widehat{\alpha} \not\prec \widehat{x} Q$	$CutL(P, \alpha, x, Q)$

This corresponds more closely to standard term graph notation, which we introduce below.

Definition 4.1.9 (Signature for $\mathcal{X}$) *The signature of the $\mathcal{X}$-calculus, is defined by the 5-tuple $\langle \mathcal{F}, \mathcal{N}, \mathcal{V}, arity, \mathcal{B} \rangle$, where:*

$$
\begin{aligned}
\mathcal{F} &= \{\mathsf{Cap}, \mathsf{Exp}, \mathsf{Imp}, \mathsf{Cut}, \mathsf{CutL}, \mathsf{CutR}\} \\
\mathcal{N} &= \{x, y, z, \ldots, \alpha, \beta, \delta, \ldots\} \\
\mathcal{V} &= \{G, H, M, N, O, P, Q, R, S, T\} \\
arity &= \{(\mathsf{Cap}, 2), (\mathsf{Exp}, 4), (\mathsf{Imp}, 5), (\mathsf{Cut}, 4), (\mathsf{CutL}, 4), (\mathsf{CutR}, 4)\} \\
\mathcal{B} &= \left\{ \begin{array}{l} \langle \mathsf{Exp}, 0, 1 \rangle, \langle \mathsf{Exp}, 2, 1 \rangle, \\ \langle \mathsf{Imp}, 1, 0 \rangle, \langle \mathsf{Imp}, 3, 4 \rangle, \\ \langle \mathsf{Cut}, 1, 0 \rangle, \langle \mathsf{Cut}, 2, 3 \rangle, \\ \langle \mathsf{CutL}, 1, 0 \rangle, \langle \mathsf{CutL}, 2, 3 \rangle, \\ \langle \mathsf{CutR}, 1, 0 \rangle, \langle \mathsf{CutR}, 2, 3 \rangle \end{array} \right\}
\end{aligned}
$$

For example, the binder specification $\langle \mathsf{Exp}, 0, 1 \rangle$ implies that the zeroth argument of the node labelled Exp binds over the first argument.

Using this signature, we can move to define an interpretation from $\mathcal{X}$-circuits to term graphs for $\mathcal{X}$. First we mention two special features of our term graphs relating to the *variable binding* and *variable identity* relations. Names (formal parameters) in our term graphs are represented as distinct node objects (rather than components of data segments of 'binder' nodes as in Wadsworth's λ-graphs, or as pointers to 'binder' nodes as in Kahl's second-order term graphs). The relation between a binder and its occurrences in the subterm it binds over is expressed using sharing: they are the same node object. Equality of free names is also expressed in this way. This sharing feature introduces an additional complexity to the interpretation of $\mathcal{X}$-circuits. In the following, we will define a recursive and one-pass interpretation function that builds a term graph $\wr P \int_n^L$ from the $\mathcal{X}$-circuit P, maintaining a list L of free names to build the sharing into the graph and a counter n of node id's.

Definition 4.1.10 (Term Graph Interpretation for $\mathcal{X}$) *For each circuit, P, we define its* term graph interpretation, *$\wr P \int_n^L$. The parameter n is a counter representing the next assignable node id. As the term graph is (inductively) constructed, a list L of sharable free variables is built up. Every interpretation of a circuit is passed such a list with which any free variables it introduces can be shared to maintain the variable identity relation.*

Anticipating the extension of the term graph interpretation to graph rewrite rules, we will define our interpretation over open rooted graphs permitting variable nodes; these graphs are described by the 7-tuple $\langle X, lab, F, N, V, \succ, z \rangle$. We will also define a function fn_g,

which will compute the set of free names and *variables in a rooted open graph g. (Notice that there are no nodes labelled as variables in term graphs).*

$$fn_g = \left\{ (i, g_{lab(i)}) \mid i \in (\texttt{fnSet}\ g_z\ g) \right\} \cup \left\{ (i, g_{lab(i)}) \mid i \in g_V \right\}$$

In addition, we define the auxiliary function `getId`*, which returns a node id for the given symbol that where possible shares nodes according to the variable binding and variable identity relations.*

$$\begin{aligned}
&\texttt{getId} :: \longrightarrow [\mathbb{N} \times (\mathcal{N} \cup \mathcal{V})] \\
&\texttt{getId}\ x\ [\,]\ n &&= n \\
&\texttt{getId}\ x\ (i, x){:}L\ n &&= i \\
&\texttt{getId}\ x\ (i, y){:}L\ n &&= \texttt{getId}\ x\ L\ n
\end{aligned}$$

The interpretation $\wr P \int_n^L$ of $\mathcal{X}$-circuits P, given below, returns a pair consisting of (i) a number to be added to the counter n which will yield the next available node id, i.e., the number of counter increments made during the interpretation of P plus one, and (ii) the term graph representation of the P.

$$\begin{aligned}
\wr \langle x{\cdot}\alpha \rangle \int_n^L \quad = \quad & \langle 3, \langle\ \{n, idx, id\alpha\}, \\
& \quad \{(n, \mathsf{Cap}), (idx, x), (id\alpha, \alpha)\}, \\
& \quad \{n\}, \\
& \quad \{idx, id\alpha\}, \\
& \quad \emptyset, \\
& \quad \{(n, [idx, id\alpha])\}, \\
& \quad n\ \rangle \\
& \rangle\ \texttt{where} \\
& \quad idx = \texttt{getId}\ x\ L\ (n{+}1) \\
& \quad id\alpha = \texttt{getId}\ \alpha\ L\ (n{+}2)
\end{aligned}$$

$$\begin{aligned}
\wr \widehat{x} P \widehat{\beta} {\cdot} \alpha \int_n^L \quad = \quad & \langle 4{+}m, \langle\ X \cup \{n, idx, id\beta, id\alpha\}, \\
& \quad lab \cup \{(n, \mathsf{Exp}), (idx, x), (id\beta, \beta), (id\alpha, \alpha)\}, \\
& \quad F \cup \{n\}, \\
& \quad N \cup \{idx, id\beta, id\alpha\}, \\
& \quad V, \\
& \quad succ \cup \{(n, [idx, z, id\beta, id\alpha])\}, \\
& \quad n\ \rangle \\
& \rangle\ \texttt{where} \\
& \qquad \langle m, PG \rangle = \wr P \int_{n+4}^{L \setminus (_, x) \setminus (_, \beta)} \\
& \quad \langle X, lab, F, N, V, succ, z \rangle = PG \\
& \qquad idx = \texttt{getId}\ x\ fn_{PG}\ (n{+}1) \\
& \qquad id\beta = \texttt{getId}\ \beta\ fn_{PG}\ (n{+}2) \\
& \qquad id\alpha = \texttt{getId}\ \alpha\ fn_{PG} \setminus (idx, _) \setminus (id\beta, _){:}L\ (n{+}3)
\end{aligned}$$

$$
\begin{array}{lcl}
\wr P \widehat{\alpha}\,[y]\,\widehat{x} Q \int_n^L & = & \langle 5{+}m_1{+}m_2, \langle\, X_1 \cup X_2 \cup \{n, id\alpha, idy, idx\}, \\
& & \quad lab_1 \cup lab_2 \cup \{(n, \mathsf{Imp}), (id\alpha, \alpha), (idy, y), (idx, x)\}, \\
& & \quad F_1 \cup F_2 \cup \{n\}, \\
& & \quad N_1 \cup N_2 \cup \{id\alpha, idy, idx\}, \\
& & \quad V_1 \cup V_2, \\
& & \quad succ_1 \cup succ_2 \cup \{(n, [z_1, id\alpha, idy, idx, z_2])\}, \\
& & \quad n \,\rangle \\
& & \rangle \texttt{ where} \\
& & \qquad \langle m_1, PG\rangle = \wr P \int_{n+4}^{L\setminus(\_,\alpha)} \\
& & \qquad \langle X_1, lab_1, F_1, N_1, V_1, succ_1, z_1\rangle = PG \\
& & \qquad\qquad L' = (fn_{PG}\setminus(id\alpha,\_){:}L)\setminus(\_, x) \\
& & \qquad \langle m_2, QG\rangle = \wr Q \int_{m_1+n+4}^{L'} \\
& & \qquad \langle X_2, lab_2, F_2, N_2, V_2, succ_2, z_2\rangle = QG \\
& & \qquad\qquad id\alpha = \texttt{getId}\,\alpha\, fn_{PG}\ (n{+}1) \\
& & \qquad\qquad idx = \texttt{getId}\, x\, fn_{QG}\ (n{+}2) \\
& & \qquad\qquad idy = \texttt{getId}\, y\, fn_{PG}\setminus(id\alpha,\_){:}fn_{QG}\setminus(idx,\_){:}L\ (n{+}3)
\end{array}
$$

$$
\begin{array}{lcl}
\wr P \widehat{\alpha} \dagger \widehat{x} Q \int_n^L & = & \langle 4{+}m_1{+}m_2, \langle\, X_1 \cup X_2 \cup \{n, id\alpha, idx\}, \\
& & \quad lab_1 \cup lab_2 \cup \{(n, \mathsf{Cut}), (id\alpha, \alpha), (idx, x)\}, \\
& & \quad F_1 \cup F_2 \cup \{n\}, \\
& & \quad N_1 \cup N_2 \cup \{id\alpha, idx\}, \\
& & \quad V_1 \cup V_2, \\
& & \quad succ_1 \cup succ_2 \cup \{(n, [z_1, id\alpha, idx, z_2])\}, \\
& & \quad n \,\rangle \\
& & \rangle \texttt{ where} \\
& & \qquad \langle m_1, PG\rangle = \wr P \int_{n+4}^{L\setminus(\_,\alpha)} \\
& & \qquad \langle X_1, lab_1, F_1, N_1, V_1, succ_1, z_1\rangle = PG \\
& & \qquad\qquad L' = (fn_{PG}\setminus(id\alpha,\_){:}L)\setminus(\_, x) \\
& & \qquad \langle m_2, QG\rangle = \wr Q \int_{m_1+n+4}^{L'} \\
& & \qquad \langle X_2, lab_2, F_2, N_2, V_2, succ_2, z_2\rangle = QG \\
& & \qquad\qquad id\alpha = \texttt{getId}\,\alpha\, fn_{PG}\ (n{+}1) \\
& & \qquad\qquad idx = \texttt{getId}\, x\, fn_{QG}\ (n{+}2)
\end{array}
$$

The interpretations $\wr P\widehat{\alpha} \nearrow \widehat{x} Q \int_n^L$ *and* $\wr P\widehat{\alpha} \nwarrow \widehat{x} Q \int_n^L$ *are the same as* $\wr P\widehat{\alpha} \dagger \widehat{x} Q \int_n^L$ *except for the use of the respective function symbols* CutL *and* CutR *(instead of* Cut*) in the specification of lab.*

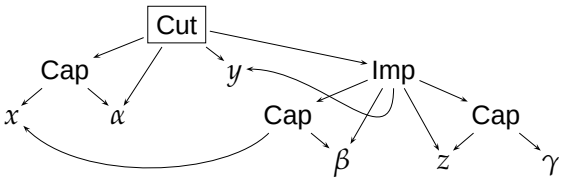
We can now define the interpretation of an arbitrary $\mathcal{X}$*-circuit* P*, as:*

$$\wr P \int = \wr P \int_0^{[\,]}$$

We note that once a circuit has been interpreted, the label of that node is not important. However, we will allow these labels since they improve the readability of the term graphs, and allow one to make direct comparisons with the original

input circuit to the interpretation.

Example 4.1.11 (An $\mathcal{X}$-circuit Interpretation) *We give an example of interpreting an $\mathcal{X}$-circuit to a* CTGRS *term graph.* $\wr\langle x{\cdot}\alpha\rangle\widehat{\alpha}\dagger\widehat{y}(\langle x{\cdot}\beta\rangle\widehat{\beta}\,[y]\,\widehat{z}\langle z{\cdot}\gamma\rangle)\int$ *becomes:*



$$\begin{array}{l}\langle\ \{0,4,5,6,7,10,11,12,14,15,16\},\\ \{(0,\mathsf{Cut}),(4,\mathsf{Cap}),(5,x),(6,\alpha),(7,\mathsf{Imp}),(10,y),\\ \ (11,\mathsf{Cap}),(12,\beta),(14,\mathsf{Cap}),(15,z),(16,\gamma)\},\\ \{0,4,7,11,14\},\\ \{5,6,10,12,15,16\},\\ \emptyset,\\ \{(0,[4,6,10,7]),(4,[5,6]),(7,[11,12,10,15,14]),\\ \ (11,[5,12]),(14,[15,16])\},\\ 0\rangle\end{array}$$

In diagrammatic representations of term graphs, we will identify root nodes with a square box as shown above.

Notice the sharing of the free names x expressing the variable identity relation, and the sharing of the formal parameters with their occurrences (i.e., the connectors y, z and β) expressing the variable binding relation.

Definition 4.1.12 (Interpretation of Reduction Rules) *The* lifting *of the reduction rules to* graph rewrite rules *is expressed by first extending the interpretation of circuits to open graphs dealing with the case of interpreting an open node (i.e., elements of $\Sigma_{\mathcal{V}}$). For the circuit variable $P \in \Sigma_{\mathcal{V}}$ we have,*

$$\begin{array}{c}\wr P\int_n^L = \langle 1, \langle\{idP\},\{(idP,P)\},\emptyset,\emptyset,\{idP\},\emptyset,n\rangle\ \rangle\\ \texttt{where}\ idP\ =\ \texttt{getId}\ P\ L\ n\end{array}$$

We need to introduce three constraints on the specification of any $\mathcal{X}$-calculus reduction rule that needs to be interpreted as term graphs; these are:

1. *The left and right-hand side of the reduction rule obeys Barendregt's convention.*
2. *All names and variables which have the same label (i.e., even bound names) are intended to be shared and will therefore be represented as a single node.*
3. *Side-conditions on the reduction rules for circuits refer only to the left-hand sides of the rule.*

We can then define the interpretation of a reduction rule as:

$$\wr left \rightarrow right \twoheadleftarrow sc\int = \langle g, z_l, z_r, sc'\rangle$$

and,

$$
\begin{aligned}
g = \texttt{share}\ \langle\ & X_l \cup X_r,\ lab_l \cup lab_r,\ F_l \cup F_r,\ N_l \cup N_r,\ V_l \cup V_r,\ succ_l \cup succ_r\ \rangle \\
& \texttt{where} \\
& \langle m_l, LG \rangle = \wr left \int_0^{[]} \\
& \langle X_l, lab_l, F_l, N_l, V_l succ_l, z_l \rangle = LG \\
& L = \left[(i, LG_{lab(i)}) \mid i \in (N_l \cup V_l) \right] \\
& \langle m_r, \langle X_r, lab_r, F_r, N_r, V_r, succ_r, z_r \rangle \rangle = \wr right \int_{m_l}^{L}
\end{aligned}
$$

and the `share` *function is defined as:*

$$
\begin{aligned}
&\texttt{share} :: \mathit{Open\ Graph} \to \mathit{Open\ Graph}_F \to \mathit{Open\ Graph}_F \to \mathit{Open\ Graph} \\
&\texttt{share}\ g\ l\ r = \langle\ \{n \in g_X \mid (\_, n) \notin \mathit{redirects}\}, \\
&\qquad \{(n, \_) \in g_{lab} \mid (\_, n) \notin \mathit{redirects}\}, \\
&\qquad g_F, \\
&\qquad \{n \in g_N \mid (\_, n) \notin \mathit{redirects}\}, \\
&\qquad g_V, \\
&\qquad \{(m, L \Leftarrow\!\!| \mathit{redirects}) \mid (m, L) \in g_{succ}\} \\
&\quad \rangle\ \texttt{where} \\
&\qquad \mathit{redirects} = \left[(m, n) \mid m \in \texttt{bnSet}\ l\ g \wedge n \in \texttt{bnSet}\ r\ g \wedge g_{lab(m)} = g_{lab(n)} \right]
\end{aligned}
$$

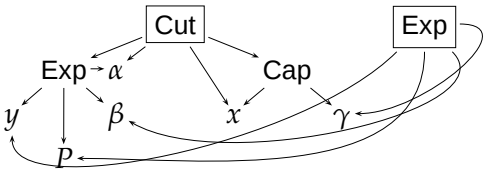
The side-conditions sc′ acting on the graph can also be mechanically interpreted by matching the labels used in sc on the $\mathcal{X}$-circuit to the (unique) node id with that label in the graph. This technique is straightforward and has been implemented in the tool; we will omit these details here. Alternatively, the conditions can be reformulated by hand to the language of graphs.

Following [14, 15], these rules induce a notion $G \to_g G'$ of term graph rewriting.

Definition 4.1.13 ($\mathcal{X}$-graphs) *We define the set of* initial *$\mathcal{X}$-graphs as the image of $\mathcal{X}$-circuits under $\wr \cdot \int$. We can then define the set of $\mathcal{X}$-graphs by closure under graph rewriting of initial $\mathcal{X}$-graphs.*

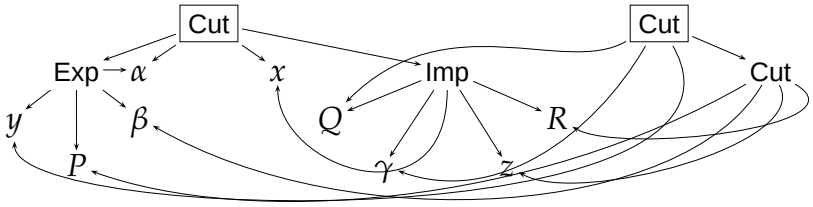
Example 4.1.14 (Example Graph Rewrite Rules) *We give some interpretations of reduction rules for the $\mathcal{X}$-calculus (Definitions 3.1.4 and 3.1.7) to* CTGRS *graph rewrite rules:*

- (*exp-rn*): $(\widehat{y} P \widehat{\beta} \cdot \alpha) \widehat{\alpha} \dagger \widehat{x} \langle x \cdot \gamma \rangle \to \widehat{y} P \widehat{\beta} \cdot \gamma \quad \leftarrow \widehat{y} P \widehat{\beta} \cdot \alpha$ *introduces* α

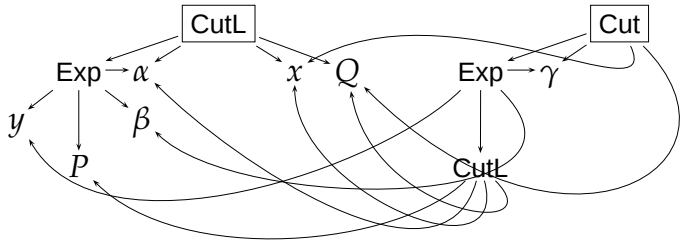


$\langle\langle\{0,4,5,6,7,8,9,10,11,12\},$
$\{(0,\mathsf{Cut}),(4,\mathsf{Exp}),(5,y),(6,\beta),(7,\alpha),$
$(8,P),(9,\mathsf{Cap}),(10,x),(11,\gamma),(12,\mathsf{Exp})\}$
$\{0,4,9,12\},$
$\{5,6,7,10,11\},$
$\{8\},$
$\{(0,[4,7,5,9]),(4,[5,6,7,8]),(9,[10,11]),$
$(12,[5,8,6,7])\}$
$\rangle\,,0,12,$ *introduces*(4,7) $\rangle$

- (*exp-imp*): $(\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R) \;\rightarrow\; Q\widehat{\gamma}\dagger\widehat{y}(P\widehat{\beta}\dagger\widehat{z}R) \;\leftarrow\; (\widehat{y}P\widehat{\beta}\cdot\alpha$ *introduces* $\alpha \wedge$ $Q\widehat{\gamma}\,[x]\,\widehat{z}R$ *introduces* $x)$



- (*exp-outs*$\not\curlywedge$): $(\widehat{y}Q\widehat{\beta}\cdot\alpha)\widehat{\alpha}\not\curlywedge\widehat{x}P \;\rightarrow\; (\widehat{y}(Q\widehat{\alpha}\not\curlywedge\widehat{x}P)\widehat{\beta}\cdot\gamma)\widehat{\gamma}\dagger\widehat{x}P$, γ *fresh*



Notice in the above graph, any right-hand side node labelled with a name which is not the label of a node reachable from left-root is automatically identified as a fresh name, represented as a unique node.

As can be seen, the amount of nodes added to the graph is small in comparison to the complexity of the graph generated by the rewriting; notice, for example, that an application of the third rule (*exp-outs*$\not\curlywedge$), although syntactically complicated, would add only the four nodes labelled Cut, Exp, γ, and CutL (that are accessible from the right root). Also, all edges coming into the node in the graph that is matched against the left-root would be redirected into the new node Cut. The node matched to CutL would become potential garbage.

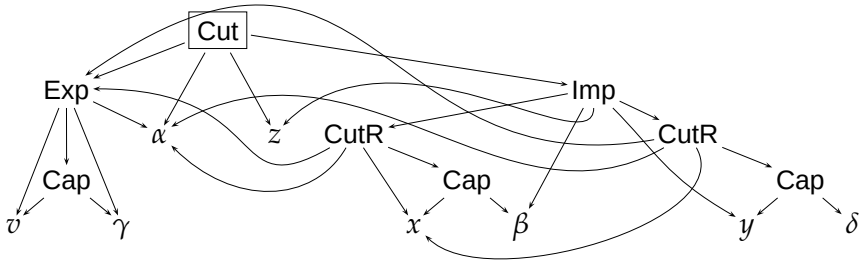
In addition to the interpretation of circuits to graphs, we would like an operation that transforms an $\mathcal{X}$-graph with sharing into one whose structure more closely resembles an $\mathcal{X}$-circuit. This is achieved by 'unravelling' the graph; copying out the shared function-symbol and metavariable nodes as far down as the connectors (which only appear once in a graph).

Definition 4.1.15 (Unravelling, c.f.,[55]) *$Unrav(G)$, the* unravelling *of an $\mathcal{X}$-graph G is obtained by traversing the (acyclic) graph top-down, and copying, for all shared graphs, all nodes in that graph that are not names.*

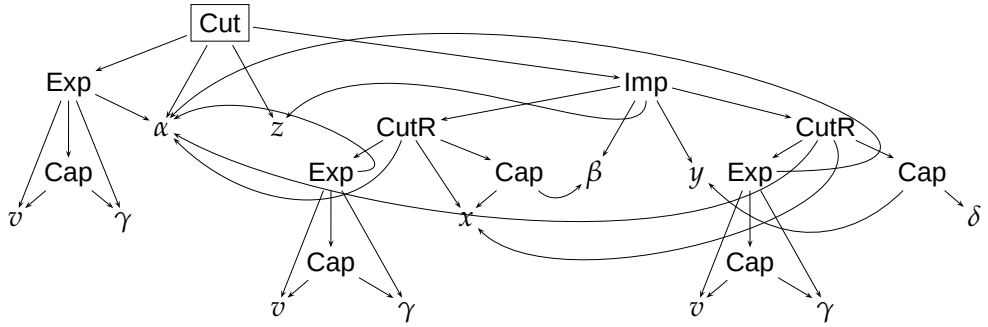
Notice that both the set of *initial* $\mathcal{X}$-graphs and the image of the set of $\mathcal{X}$-graphs under $Unrav(\cdot)$ are graphs containing sharing only at the level of connectors. This setup gives us a method of comparing an $\mathcal{X}$-circuit P with an $\mathcal{X}$-graph G, by comparing $\wr P \int$ with $Unrav(G)$. This will be useful for formulating results later in the paper.

Example 4.1.16 (Unravelling of an $\mathcal{X}$-Graph) *Take G to be*

$$\wr(\widehat{v}\langle v\cdot\gamma\rangle\widehat{\gamma}\cdot\alpha)\widehat{\alpha} \dagger \widehat{z}(((\widehat{v}\langle v\cdot\gamma\rangle\widehat{\gamma}\cdot\alpha)\widehat{\alpha} \nwarrow \widehat{x}\langle x\cdot\beta\rangle)\widehat{\beta}\,[z]\,\widehat{y}((\widehat{v}\langle v\cdot\gamma\rangle\widehat{\gamma}\cdot\alpha)\widehat{\alpha} \nwarrow \widehat{x}\langle y\cdot\alpha\rangle))\int$$



then $Unrav(G)$ is



Notice that the bound connectors v and γ within the shared graph $\mathsf{Exp}(1{:}v, \mathsf{Cap}(1,2), 2{:}\gamma, \alpha)$ are copied out, but α is not, and that the in-degree of α increases.

We now have the following results.

Lemma 4.1.17 *If $G_1 \rightarrow_g G_2$, then there exists G_3 such that $Unrav(G_1) \rightarrow_g G_3$, as well as $Unrav(G_2) = Unrav(G_3)$.*

Proof 4.1.18 *In each step of $G_1 \rightarrow_g G_2$ a cut K is contracted. Using colouring, we can build a reduction sequence $Unrav(G_1) \rightarrow_g G_3$, for some G_3, by contracting, for each step*

in $G_1 \rightarrow_g G_2$, always only all copies of K (using the same rule repeatedly). Notice that this reduction might have introduced sharing, and that G_2 and G_3 differ only in that G_3 contains less sharing than G_2, i.e. G_3 is a partially unravelled version of G_2. Since no other manipulation has been performed, we get $\mathsf{Unrav}(G_2) = \mathsf{Unrav}(G_3)$. □

We also have the following adequacy result:

Theorem 4.1.19 (Adequacy) *Let G_1, G_2 be $\mathcal{X}$-graphs, and P_1, P_2 be $\mathcal{X}$-circuits such that $\mathsf{Unrav}(G_i) = \wr P_i \int$, for $i = 1, 2$. If $G_1 \rightarrow_g G_2$, then $P_1 \rightarrow P_2$. Moreover, if G_2 is in normal form, then so is P_2.*

Proof 4.1.20 *By Lemma 4.1.17, we get that there exists a G_3 such that $\wr P_1 \int \rightarrow_g G_3$, as well as $\wr P_2 \int = \mathsf{Unrav}(G_3)$. This reduction induces, similar to Lemma 4.1.17, a reduction from P_1 to P_2. If G_2 contains no cuts, then neither does $\wr P_2 \int$, nor P_2.* □

We can now prove the following result:

Theorem 4.1.21 *If $P \rightarrow Q$ in one step, then there exists a $\mathcal{X}$-graph G such that: $\wr P \int \rightarrow_g G$, and $\mathsf{Unrav}(G) = \wr Q \int$.*

Proof 4.1.22 *Easy; in $\wr P \int$, redexes are not shared; the only sharing in G is introduced by the reduction, which gets erased by unravelling.* □

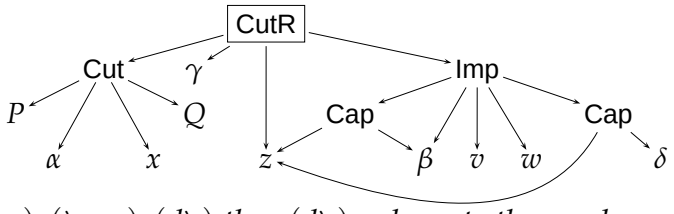
Notice that, by the non-confluent character for $\mathcal{X}$, we cannot prove a similar result for many-steps reduction paths, as illustrated by the following example.

Example 4.1.23 (Sharing and Non-confluence) *Let P and Q be such that $\alpha \notin fp(P)$ and $x \notin fp(Q)$, so $P \leftarrow P\widehat{\alpha} \dagger \widehat{x}Q \rightarrow Q$. Now (assume $z \neq v$):*

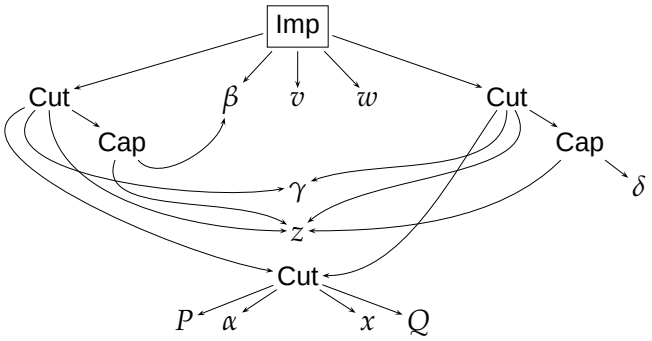
$$
\begin{array}{ll}
(P\widehat{\alpha} \dagger \widehat{x}Q)\widehat{\gamma} \nwarrow \widehat{z}(\langle \boldsymbol{z}\cdot\beta\rangle \widehat{\beta}\,[v]\, \widehat{w}\langle \boldsymbol{z}\cdot\delta\rangle) & \rightarrow (\nwarrow imp\text{-}ins), (d\nwarrow), (d\nwarrow) \\
((\boldsymbol{P\widehat{\alpha}} \dagger \widehat{\boldsymbol{x}}\boldsymbol{Q})\widehat{\gamma} \dagger \widehat{z}\langle z\cdot\beta\rangle)\widehat{\beta}\,[v]\, \widehat{w}((\boldsymbol{P\widehat{\alpha}} \dagger \widehat{\boldsymbol{x}}\boldsymbol{Q})\widehat{\gamma} \dagger \widehat{z}\langle z\cdot\delta\rangle) & \rightarrow (act\text{-}\mathrm{L}), (\not\nwarrow gc), (act\text{-}\mathrm{R}), (\nwarrow gc) \\
(\boldsymbol{P}\widehat{\gamma} \dagger \widehat{z}\langle z\cdot\beta\rangle)\widehat{\beta}\,[v]\, \widehat{w}(\boldsymbol{Q}\widehat{\gamma} \dagger \widehat{z}\langle z\cdot\delta\rangle) &
\end{array}
$$

Notice that we have explicitly used the non-confluence of the cut $P\widehat{\alpha} \dagger \widehat{x}Q$, and reduced it once to P, and once to Q.

We cannot simulate this in the setting of term graph rewriting. Instead, we get the following graph for the first term,



which, by $(\curlywedge imp\text{-}ins)$, $(\curlywedge exp)$, $(d\curlywedge)$ *then* $(d\curlywedge)$ *reduces to the graph,*



Since the cut $\mathsf{Cut}(P, \alpha, x, Q)$ *is shared, it can only be reduced* once, *resulting in either* P *or* Q *as the common subterm to the respective parent cuts. This implies the previously illustrated reduction cannot be simulated.*

This is not unexpected, however, since all implementations of reduction systems will use a *reduction strategy*, preferring certain redexes over others, and thereby excluding other reduction paths.

On the other hand, when restricting to either the CBN or CBV-reduction strategies, the above negative result does not hold; in fact, we can show that for confluent reduction, our term-graph rewriting engine models reduction in $\mathcal{X}$.

4.2 Name Capture and Clash in $\mathcal{X}$

So far in this chapter, we have built a system which is expressive enough to describe the syntax and reduction rules of the $\mathcal{X}$-calculus. We specified how to formulate side-conditions on the graph rewrite rules, and how to express the higher-order variable binding and variable identity relations.

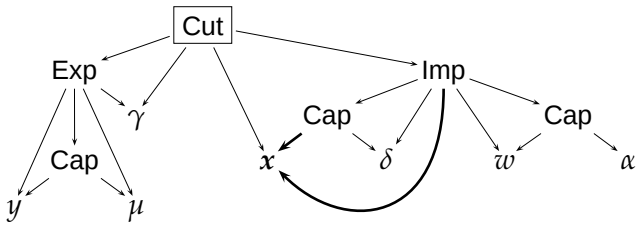
As discussed in the background section on rewriting (Section 2.4), simply defining these higher-order relations is not enough—a means of maintaining them must also be specified for reductions to be correct. In the following example we will highlight the problems (specifically incorrect reduction sequences) which are

the result of not maintaining the binding relations. We will illustrate this using the $\mathcal{X}$-graphs defined in the previous section.

Example 4.2.1 (Name Clash in $\mathcal{X}$) *We will highlight portions of circuits (using dashed lines, bold lines and bold symbols) to guide the reader through an example reduction, illustrating name clash. Take the following circuit,*

$$(\widehat{y}\langle y{\cdot}\mu\rangle\widehat{\mu}{\cdot}\gamma)\widehat{\gamma}\dagger\widehat{x}(\langle\boldsymbol{x}{\cdot}\delta\rangle\widehat{\delta}\,[\boldsymbol{x}]\,\widehat{w}\langle w{\cdot}\alpha\rangle)$$

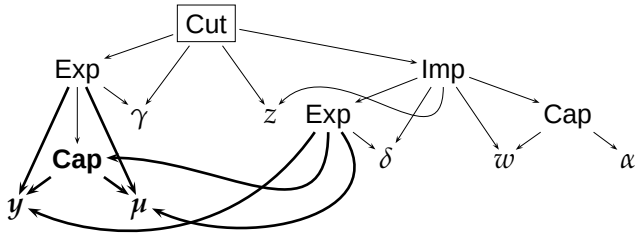
This circuit corresponds to the $\lambda\boldsymbol{x}$-term $x\boldsymbol{x}\,\langle x = \lambda y.y\rangle$; notice that reducing this $\lambda\boldsymbol{x}$-term poses no name capture problem. The $\mathcal{X}$-graph that represents the above circuit (built using Definition 4.1.10) is:



(4.1)

Applying the term graph rules (act-R), (↖imp-outs), (d↖), (exp-rn), then (↖cap) will generate:

$$(\widehat{\boldsymbol{y}}\langle\boldsymbol{y}{\cdot}\boldsymbol{\mu}\rangle\widehat{\boldsymbol{\mu}}{\cdot}\gamma)\widehat{\gamma}\dagger\widehat{z}((\widehat{\boldsymbol{y}}\langle\boldsymbol{y}{\cdot}\boldsymbol{\mu}\rangle\widehat{\boldsymbol{\mu}}{\cdot}\boldsymbol{\delta})\widehat{\delta}\,[z]\,\widehat{w}\langle w{\cdot}\alpha\rangle)$$

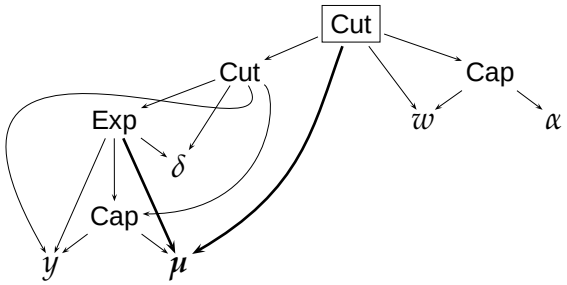


(4.2)

with z fresh.

As is clear from this graph, the capsule on the left is now shared. Also, there are two *binders to both y and μ, coming from the two export nodes. Continuing the execution of*

this graph via $(\textit{exp-imp}_{cbn})$ *yields the graph,* $((\widehat{y}\langle y\cdot\mu\rangle\widehat{\boldsymbol{\mu}}\cdot\delta)\widehat{\delta}\dagger\widehat{y}\langle y\cdot\mu\rangle)\widehat{\boldsymbol{\mu}}\dagger\widehat{w}\langle w\cdot\alpha\rangle$



(4.3)

Notice that now there are two nested *binders to* μ*: one coming from* Exp*, the second coming from the top-most* Cut*.*

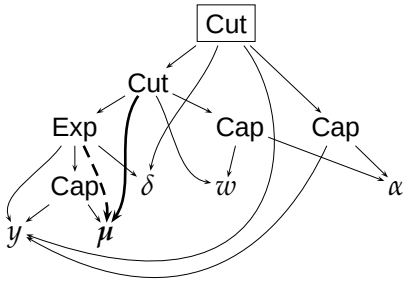
According to Lemma 3.1.25, the outermost cut $\widehat{\mu}\dagger\widehat{w}$ *should behave as a* renaming *cut, renaming all free occurrences of* μ *in the term by* α*, resulting in,*

$$(\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\delta)\widehat{\delta}\dagger\widehat{y}\langle y\cdot\boldsymbol{\alpha}\rangle \qquad (\star)$$

However, propagating this cut through the left circuit in a stepwise fashion, we are presented with an incorrect reduction step.

$$\begin{array}{ll}((\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\delta)\widehat{\delta}\dagger\widehat{y}\langle y\cdot\mu\rangle)\widehat{\boldsymbol{\mu}}\dagger\widehat{w}\langle w\cdot\alpha\rangle & \longrightarrow (\textit{act-}\mathrm{L}),(\textit{cut}\not\star)\\ ((\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\delta)\widehat{\boldsymbol{\mu}}\not\star\widehat{w}\langle w\cdot\alpha\rangle)\widehat{\delta}\dagger\widehat{y}(\langle y\cdot\mu\rangle\widehat{\boldsymbol{\mu}}\not\star\widehat{w}\langle w\cdot\alpha\rangle) & \longrightarrow (\not\star d),(\textit{cap-rn}),(\not\star d)\\ ((\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\delta)\widehat{\boldsymbol{\mu}}\dagger\widehat{w}\langle w\cdot\alpha\rangle)\widehat{\delta}\dagger\widehat{y}\langle y\cdot\alpha\rangle & \end{array}$$

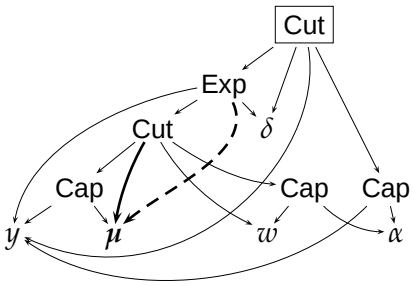
The $\mathcal{X}$*-graph for this last circuit is:*



(4.4)

The generated graph shares all occurrences of the μ *connector in the circuit. The only rule applicable in this case is* $(\textit{exp-ins}\not\star)$ *(since* μ *is not introduced in the corresponding circuit), and as displayed by the graph below, the rule causes the scopes of the two* μ *binders (used by the cut and the export) to swap.*

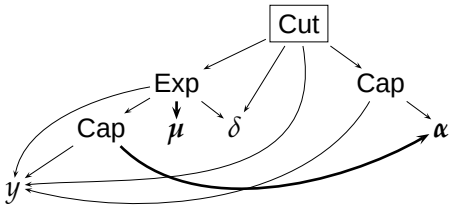
$$(\widehat{y}(\langle y\cdot\mu\rangle\widehat{\boldsymbol{\mu}}\dagger\widehat{w}\langle w\cdot\alpha\rangle)\widehat{\mu}\cdot\delta)\widehat{\delta}\dagger\widehat{y}\langle y\cdot\alpha\rangle$$



(4.5)

The cut now deactivates and incorrectly renames the μ in the capsule to α, which destroys the relations between the body of the original export, $\langle y{\cdot}\mu\rangle$, and its formal parameters y and μ. The reduced term is,

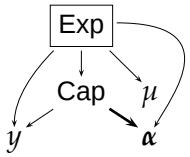
$$(\widehat{y}\langle y{\cdot}\boldsymbol{\alpha}\rangle\widehat{\boldsymbol{\mu}}{\cdot}\delta)\,\widehat{\delta}\dagger\widehat{y}\langle y{\cdot}\alpha\rangle$$



(4.6)

Compare how the above circuit differs from circuit (⋆)*; notice that the body of the export now sends it output to α, rather than μ. This differs from the expected term in that the innermost μ of circuit* (⋆) *(above), has been renamed as well. The term finally reduces by* (*exp-rn*) *to,*

$$\widehat{y}\langle y{\cdot}\boldsymbol{\alpha}\rangle\widehat{\mu}{\cdot}\alpha$$



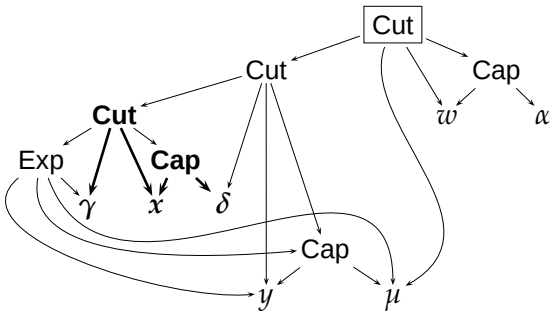
(4.7)

Example 4.2.2 (Name Capture in $\mathcal{X}$) *In this example, we will illustrate the problem of name capture. We begin with circuit* (4.1)*:*

$$(\widehat{y}\langle y{\cdot}\mu\rangle\widehat{\mu}{\cdot}\gamma)\,\widehat{\gamma}\dagger\widehat{x}(\langle x{\cdot}\delta\rangle\widehat{\delta}\,[x]\,\widehat{w}\langle w{\cdot}\alpha\rangle)$$

However, instead this time we apply only the rules (*act*-R), (↖*imp-outs*), (↖*cap*), (*d*↖). *This is followed by an application of* (*exp-imp*$_{cbn}$), *giving the circuit:*

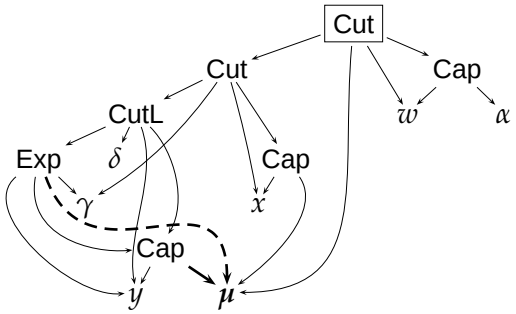
$$(((\widehat{y}\langle y{\cdot}\mu\rangle\widehat{\mu}{\cdot}\boldsymbol{\gamma})\,\widehat{\boldsymbol{\gamma}}\dagger\widehat{\boldsymbol{x}}\langle\boldsymbol{x}{\cdot}\boldsymbol{\delta}\rangle)\,\widehat{\delta}\dagger\widehat{y}\langle y{\cdot}\mu\rangle)\,\widehat{\mu}\dagger\widehat{w}\langle w{\cdot}\alpha\rangle$$



(4.8)

In contrast to Circuit (4.3)*, there is an extra renaming cut* $\widehat{\gamma} \dagger \widehat{x}$ *with an export (highlighted). The presence of this cut allows for the left-activation of the cut* $\widehat{\delta} \dagger \widehat{y}$*, followed by its propagation through the left sub-circuit. This is done by applying the rules* (*act*-L), (*cut*↛), (*d*↖), (*cap-rn*)*, giving us the circuit:*

$$(((\widehat{y}\langle y \cdot \mu \rangle \widehat{\boldsymbol{\mu}} \cdot \gamma)\widehat{\delta} \not\prec \widehat{y}\langle y \cdot \boldsymbol{\mu} \rangle)\widehat{\gamma} \dagger \widehat{x}\langle x \cdot \mu \rangle)\widehat{\mu} \dagger \widehat{w}\langle w \cdot \alpha \rangle$$

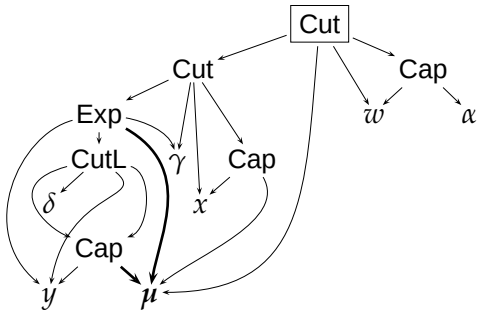


(4.9)

Notice that in the redex $(\widehat{y}\langle y \cdot \mu \rangle \widehat{\boldsymbol{\mu}} \cdot \gamma)\widehat{\delta} \not\prec \widehat{y}\langle y \cdot \boldsymbol{\mu} \rangle$*, the* μ *connector is* both *free and bound.*

By applying the rule (*exp-ins*↛)*, the free* μ *of the capsule gets* captured *by the bound* μ *of the export. We get the graph:*

$$(((\widehat{y}(\langle y \cdot \boldsymbol{\mu} \rangle \widehat{\delta} \not\prec \widehat{y}\langle y \cdot \boldsymbol{\mu} \rangle)\widehat{\boldsymbol{\mu}} \cdot \gamma)\widehat{\gamma} \dagger \widehat{x}\langle x \cdot \mu \rangle)\widehat{\mu} \dagger \widehat{w}\langle w \cdot \alpha \rangle$$



(4.10)

Notice in the circuit the highlighted μ *is now bound by the export, rather than the cut*

$\widehat{\mu} \dagger \widehat{w}$.

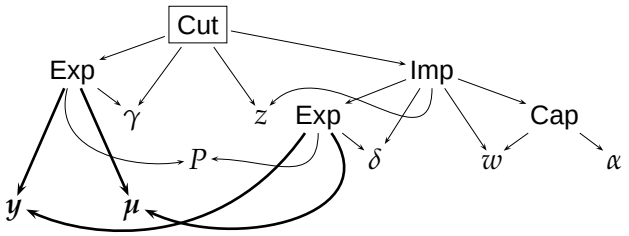
Although in this case, the capture of the free μ connector does not lead to an incorrect result (the propagating cut with that connector is destroyed), there are (more complicated and involved) examples where the result is affected. However, for the purposes of the example, we have highlighted the situation we will refer to as the name capture.

4.2.1 Lazy Copying of Shared Graphs

The solution to the problem of capture we propose in this section is to avoid, as for *λ-graphs*, the sharing of graphs that are involved in a a redex, i.e., forbid the sharing of binders involved in cuts. Similarly to the case for the λ-calculus [92], binding of connectors can be considered problematic in the context of sharing. Sharing an abstraction $\lambda x.G$ in *λ-graphs* is problematic since the substitution is implemented via a redirection on G. This can be done only once, blocking a reuse of a shared abstraction, that therefore has to be copied first. To tackle this problem within the context of $\mathcal{X}$, a notion of *rebinding of sockets* and *of plugs* was introduced.

The basic idea is the following: suppose we are dealing with the $\mathcal{X}$-graph (which can be generated by the graph rewrite system as shown in Example 4.2.1),

$$(\widehat{\boldsymbol{y}} P \widehat{\boldsymbol{\mu}} \cdot \gamma) \widehat{\gamma} \dagger \widehat{z} ((\widehat{\boldsymbol{y}} P \widehat{\boldsymbol{\mu}} \cdot \delta) \widehat{\delta}\, [z]\, \widehat{w} \langle w \cdot \alpha \rangle)$$



The fact that y and μ is bound *twice* and shared might cause the binders to later come into a position where they can interact with each other during the reductions. We avoid that by copying the parts of P that depend on y or μ: we will 'peel off a copy' of the graph which might get affected by the double binding of connectors.

This is similar to Wadsworth's notion of R-admissibility (Definition 2.4.9), and differs in that in creating an 'R-admissible' graph, we must copy several constructors and consider two classes of connectors. Unlike Wadsworth's technique

however, we noticed that eager copying of graphs will destroy a large amount of sharing. We will later specify a lazy *reduction strategy* that avoids much of the unnecessary copying (see Section 4.3.4).

First we will deal with making $\mathcal{X}$-graphs, or rather $\mathcal{X}$-subgraphs, identified as redexes 'R-admissible'. To this end, extend the signature Σ of $\mathcal{X}$-graphs with two higher-order function symbols.

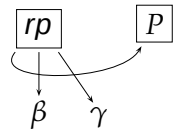
$$\begin{aligned}\Sigma' = \langle & \Sigma_{\mathcal{F}} \cup \{\mathsf{rp}, \mathsf{rs}\} \\ & \Sigma_{\mathcal{N}} \\ & \Sigma_{\mathcal{V}} \\ & \Sigma_{arity} \cup \{(\mathsf{rp}, 3), (\mathsf{rs}, 3)\} \\ & \Sigma_{\mathcal{B}} \cup \{(\mathsf{rp}, 1, 0), (\mathsf{rs}, 1, 0)\} \\ \rangle &\end{aligned}$$

The function symbols rp and rs will represent the renaming of a bound (rebinding) plug or socket, respectively. This results in the (term graph) definition of rebinding a socket (rp) as given in Definition 4.2.3. These will be used to prevent a connector from being doubly bound by, essentially, copying that structure of a graph which contains that binder whilst introducing the new connector, thereby destroying the sharing of the connector via binding edges.

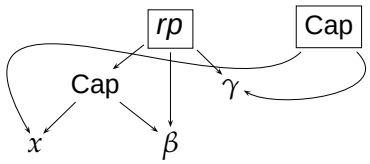
The term $\mathsf{rp}(P, \alpha, \beta)$ as given in Definition 4.2.3 is defined to build a new graph G' where the free occurrences of α in G are replaced with β and any binders encountered *in* G are made fresh. Since this is, essentially, a copying function, when we move the rebinding mechanism *under* binders, as in the third case below, we would create double binders for those bound connectors we have just passed. Therefore, we need to rebind those as well.

Definition 4.2.3 (Rebinding Rewrite Rules) *The function* rp *is defined by the following term graph rewriting rules:*

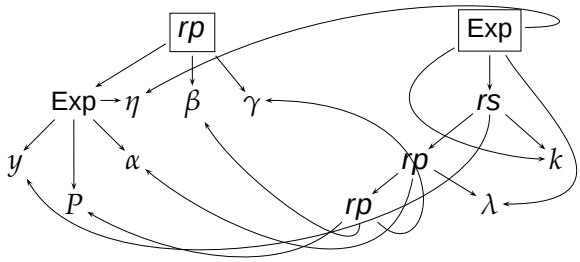
1. *(rpGC):* $\mathsf{rp}(P, \beta, \gamma) \to P \quad \leftarrow \beta \notin fc(P)$



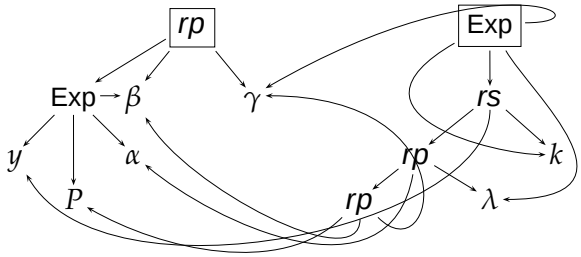
2. *(rpCapRename):* $\mathsf{rp}(\langle x \cdot \beta \rangle, \beta, \gamma) \to \langle x \cdot \gamma \rangle$



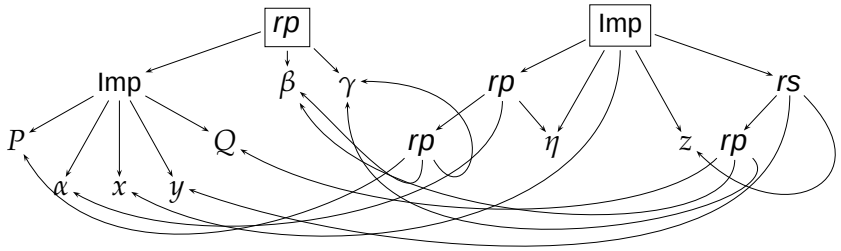
3. *(rpExp):* $\mathsf{rp}(\widehat{y} P \widehat{\alpha} \cdot \eta, \beta, \gamma) \to \widehat{k}\, \mathsf{rs}(\mathsf{rp}(\mathsf{rp}(P, \beta, \gamma), \alpha, \lambda), y, k)\ \widehat{\lambda} \cdot \eta \leftarrow \eta \neq \beta$



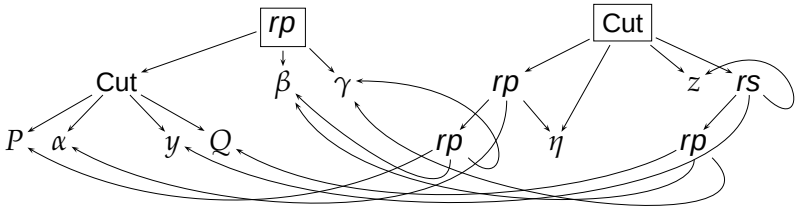
4. *(rpExpRename):* $\mathsf{rp}(\widehat{y} P \widehat{\alpha} \cdot \beta, \beta, \gamma) \to \widehat{k}\, \mathsf{rs}(\mathsf{rp}(\mathsf{rp}(P, \beta, \gamma), \alpha, \lambda), y, k)\ \widehat{\lambda} \cdot \gamma$



5. *(rpMed):* $\mathsf{rp}(P \widehat{\alpha}\ [x]\ \widehat{y} Q, \beta, \gamma) \to \mathsf{rp}(\mathsf{rp}(P, \beta, \gamma), \alpha, \eta)\ \widehat{\eta}\ [x]\ \widehat{z}\, \mathsf{rs}(\mathsf{rp}(Q, \beta, \gamma), y, z)$



6. *(rpCut):* $\mathsf{rp}(P \widehat{\alpha} \dagger \widehat{y} Q, \beta, \gamma) \to \mathsf{rp}(\mathsf{rp}(P, \beta, \gamma), \alpha, \eta)\ \widehat{\eta} \dagger \widehat{z}\, \mathsf{rs}(\mathsf{rp}(Q, \beta, \gamma), y, z)$



(The function rs *is defined similarly.) Notice that the call to the function* rp *builds an α-equivalent version of P that uses a fresh socket γ to connect rather than β. Also, all bound connectors are renamed: evaluating the rebinding rules builds a version of P with fresh binder names. This ensures there is only ever one pointer to nodes that bind over P or the local binders in P.*

The functions rs and rp are expressed as higher-order term-graph rewriting rules. Because these higher-order functions may not necessarily be evaluated eagerly, they may interfere with the reductions of the $\mathcal{X}$-calculus: if the sub-circuit of an inactive cut is a rebinding term, an activation will be forced even though the sub-circuit of the rebinding term introduces the appropriate connector of the cut (and a logical rule should therefore have be applied).

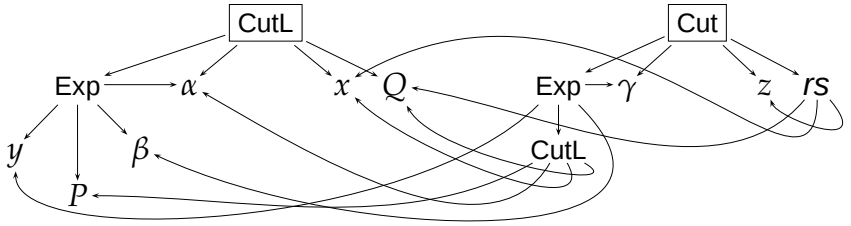
Rather than forcing the evaluation of these rebinding constructs to completion via an 'eager' reduction strategy, we will give define a lazier evaluation strategy that avoids this mis-activation in Section 4.3.4.

Using the functions rs and rp gives a different formal definition of interpreting rewrite rules in $\mathcal{X}$ as graphs. The term graph representation of each rule needs to be revised to ensure binders are not shared, resulting in the new rules that are quite involved. As suggested by the example above (Example 4.2.1), term rewrite rules which introduce sharing of binders need to copy these out these in order to avoid capture.

Definition 4.2.4 (Copying TGRS Rewrite Rules) *There are six rules of the $\mathcal{X}$-calculus (incidentally all propagation rules) which need to be modified to ensure binders are not shared. We give these modified rules below.*

Left propagation

$$(exp\text{-}outs\not\prec) : (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha} \not\prec \widehat{x}Q \rightarrow (\widehat{y}(P\widehat{\alpha} \not\prec \widehat{x}Q)\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}\,\mathsf{rs}(Q,x,z)$$

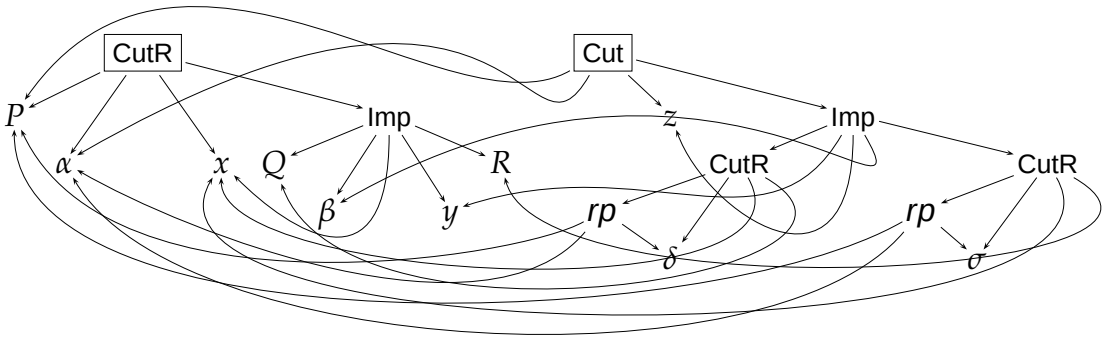


$$(imp\not\prec) : (Q\widehat{\beta}\,[z]\,\widehat{y}R)\widehat{\alpha} \not\prec \widehat{x}P \rightarrow (Q\widehat{\alpha} \not\prec \widehat{x}P)\widehat{\beta}\,[z]\,\widehat{y}(R\widehat{\alpha} \not\prec \widehat{k}\,\mathsf{rs}(P,x,k))$$

$$(cut\not\prec) : (Q\widehat{\beta} \dagger \widehat{y}R)\widehat{\alpha} \not\prec \widehat{x}P \rightarrow (Q\widehat{\alpha} \not\prec \widehat{x}P)\widehat{\beta} \dagger \widehat{y}(R\widehat{\alpha} \not\prec \widehat{k}\,\mathsf{rs}(P,x,k))$$

Right propagation

$$(\nwarrow\!\mathit{imp\text{-}outs}) : P\widehat{\alpha} \nwarrow \widehat{x}(Q\widehat{\beta}\,[x]\,\widehat{y}R) \;\to\; P\widehat{\alpha} \dagger \widehat{z}((\mathsf{rp}(P,\alpha,\mu)\;\widehat{\mu} \nwarrow \widehat{x}Q)\widehat{\beta}\,[z]\,\widehat{y}(\mathsf{rp}(P,\alpha,\eta)\;\widehat{\eta} \nwarrow \widehat{x}R))$$



$$(\nwarrow\!\mathit{imp\text{-}ins}) : P\widehat{\alpha} \nwarrow \widehat{x}(Q\widehat{\beta}\,[z]\,\widehat{y}R) \;\to\; (P\widehat{\alpha} \nwarrow \widehat{x}Q)\widehat{\beta}\,[z]\,\widehat{y}(\mathsf{rp}(P,\alpha,\gamma)\;\widehat{\gamma} \nwarrow \widehat{x}R)$$

$$(\nwarrow\!\mathit{cut}) : P\widehat{\alpha} \nwarrow \widehat{x}(Q\widehat{\beta} \dagger \widehat{y}R) \;\to\; (P\widehat{\alpha} \nwarrow \widehat{x}Q)\widehat{\beta} \dagger \widehat{y}(\mathsf{rp}(P,\alpha,\gamma)\;\widehat{\gamma} \nwarrow \widehat{x}R)$$

It is perhaps not obvious that this (partial) copy action gives a solution to the name capture problem. But, since the interpretation of circuits to term graphs (Definition 4.1.10) ensures binders are not shared, this property is preserved during reduction. So it is impossible for names to be captured.

Example 4.2.5 *A corrected reduction, using rebinding, for that of Example 4.2.1 becomes:*

$$\begin{aligned}
&(\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\gamma)\widehat{\gamma} \dagger \widehat{x}(\langle \boldsymbol{x}\cdot\delta\rangle\widehat{\delta}\,[\boldsymbol{x}]\,\widehat{w}\langle w\cdot\alpha\rangle)\\
\to\; &(\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}((\mathsf{rp}(\widehat{\boldsymbol{y}}\langle \boldsymbol{y}\cdot\boldsymbol{\mu}\rangle\widehat{\boldsymbol{\mu}}\cdot\boldsymbol{\gamma},\boldsymbol{\gamma},\boldsymbol{\tau})\widehat{\tau} \nwarrow \widehat{x}\langle x\cdot\delta\rangle)\widehat{\beta}[z]\widehat{y}\,(\mathsf{rp}(\widehat{\boldsymbol{y}}\langle \boldsymbol{y}\cdot\boldsymbol{\mu}\rangle\widehat{\boldsymbol{\mu}}\cdot\boldsymbol{\gamma},\boldsymbol{\gamma},\boldsymbol{\sigma})\widehat{\sigma} \nwarrow \widehat{x}\langle w\cdot\alpha\rangle)))\\
\to\; &(\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}(((\widehat{\boldsymbol{v}}\langle \boldsymbol{v}\cdot\boldsymbol{\nu}\rangle\widehat{\boldsymbol{\nu}}\cdot\boldsymbol{\tau})\widehat{\tau} \nwarrow \widehat{x}\langle x\cdot\delta\rangle)\widehat{\delta}\,[z]\,\widehat{w}((\widehat{\boldsymbol{u}}\langle \boldsymbol{u}\cdot\boldsymbol{\eta}\rangle\widehat{\boldsymbol{\eta}}\cdot\boldsymbol{\sigma})\widehat{\sigma} \nwarrow \widehat{x}\langle w\cdot\alpha\rangle)))\\
\to\; &(\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}((\widehat{v}\langle v\cdot\nu\rangle\widehat{\nu}\cdot\delta)\widehat{\delta}\,[z]\,\widehat{w}\langle w\cdot\alpha\rangle)\\
\to\; &((\widehat{v}\langle v\cdot\nu\rangle\widehat{\nu}\cdot\delta)\widehat{\delta} \dagger \widehat{y}\langle y\cdot\mu\rangle)\widehat{\boldsymbol{\mu}} \dagger \widehat{\boldsymbol{w}}\langle w\cdot\alpha\rangle
\end{aligned}$$

Notice that this time there is no possibility of variable clash, since there are no shared binders (the other copy of the μ binder together with the μ in the capsule is renamed to ν). The highlighted cut $\widehat{\mu} \dagger \widehat{w}$ can be activated and safely propagated through the left sub-circuit, which after an additional renaming results in:

$$\widehat{v}\langle v\cdot\nu\rangle\widehat{\nu}\cdot\alpha$$

The solution using rebinding is surprisingly easy to formulate, and only the rules that use explicit replication need to be changed, but comes at the price of having to extend the signature of the calculus, as well as the set of rewrite rules. Moreover, it turns out to be highly inefficient; this is of course mainly due to the loss of sharing. The main objection to rebinding is that it creates unnecessary overhead

in that it invokes rebinding for all double bindings of connectors, regardless of whether or not they created a conflict; as we will see in the benchmarks section (Section 4.3), the cost of running rebinding is high.

4.2.2 Preserving Barendregt's convention

Although the solution obtained by rigourous copying of shared graphs is correct in that it avoids the creation of shared binders, it was noticed that the fact that a graph can share binders is not necessarily problematic. It can be shown that some reduction paths which, although allow binders to be shared, do not lead to incorrect results. In addition, notice that in Example 4.2.1, the sharing of the y binder was never a problem. The conclusion of this observation was that, unlike for λ-graphs where the sharing of abstractions in the graphs created the problem, the problem here is of a different nature.

It is common practice to say that α-conversion is the machinery necessary to uphold Barendregt's convention. Barendregt's convention states that an identifier should not appear both free and bound in a context (where a context can be a term, but also a type statement) [13, Convention 2.1.13]. It is especially the notion of *binding* that is important; for example, normally x is considered bound in all $\lambda x.M$, $M\langle x = N\rangle$, and $\Gamma, x{:}A \vdash_\lambda M : B$. In this section, we will propose a solution to maintain the variable binding relations during reductions in the $\mathcal{X}$, by preserving Barendregt's convention on names, i.e. make sure that names never occur both free and bound. We will do this by detecting and avoiding it, without having to extend the signature of the calculus, but by modifying the rules and their side-conditions.

To tackle it in a formal way, we first introduce the notion of α-safety.

Definition 4.2.6 (α-safety) *We call a* circuit *($\mathcal{X}$-graph) α-*safe *if it adheres to Barendregt's convention, i.e. no connector occurs both free and bound, and no nesting of binders to the same connector occurs. We call a* rewrite rule *α-*safe *if it respects α-safety, that is, it rewrites an α-safe circuit (graph) to an α-safe circuit (graph). We call a rewrite system α-safe if all its rules are α-safe.*

For example, the circuit $(\widehat{y}\langle y{\cdot}\mu\rangle\widehat{\mu}{\cdot}\mu)\widehat{\mu} \dagger \widehat{w}\langle w{\cdot}\alpha\rangle$ is not α-safe (it fails both criteria); neither is $(\widehat{y}\langle y{\cdot}\mu\rangle\widehat{\mu}{\cdot}\delta)\widehat{\mu} \dagger \widehat{w}\langle w{\cdot}\alpha\rangle$, by the second criterion.

In order to obtain an α-safe implementation of $\mathcal{X}$, we need to identify the rewrite rules that are *not* α-safe. In Example 4.2.1, the application of $(\textit{exp-imp}_{cbn})$ in the third graph violates our α-safety property since μ is both bound and free in the sub-circuit $(\widehat{y}\langle y\cdot\mu\rangle\widehat{\boldsymbol{\mu}}\cdot\delta)\widehat{\delta}\,\dagger\,\widehat{y}\langle y\cdot\boldsymbol{\mu}\rangle$. So the rule $(\textit{exp-imp}_{cbn})$ is *not* α-safe. In fact, neither is the rule $(\textit{exp-imp}_{cbv})$.

We can systematically check that $(\textit{exp-imp}_{cbv})$ (which we recall below) is not α-safe by checking whether the described transformation violates the criteria. We recall the rule from Definition 3.1.4 below.

$$(\textit{exp-imp}_{cbn})\colon\ (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\,\dagger\,\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R)\ \rightarrow\ (Q\widehat{\gamma}\,\dagger\,\widehat{y}P)\widehat{\beta}\,\dagger\,\widehat{z}R\ \leftarrow\ \alpha, x\ \textit{introduced}$$

If we assume the α-safety criteria holds on an instance of the left-hand side of the rule (so $\gamma \notin bp(\widehat{y}P\widehat{\beta}\cdot\alpha)$ and $x \notin bs(Q\widehat{\gamma}\,[x]\,\widehat{z}R)$), then the application of the rule should not break the α-safety criterion. That is, we require the right-hand side of the rule to also be α-safe. Notice that in its current form, the rule does indeed break the criteria, since β and γ are nested on the right-hand side and the left-hand side places no constraints on the relation between β and γ. A term graph on which this rule is applied may have $\beta{=}\gamma$, in which case the application of the rule will have created a nested binding.

Since now we do not necessarily need to avoid connectors being bound twice (as long as they are not nested), we do not need to completely copy circuits. Instead, in dealing with the necessary renaming of bound connectors we can take advantage of the explicit renaming feature of $\mathcal{X}$, introducing to the rules new cuts such as $\langle v\cdot\delta\rangle\widehat{\delta}\,\text{⅄}\,\widehat{y}P$ or $P\widehat{\beta}\,\text{⅄}\,\widehat{v}\langle v\cdot\delta\rangle$ to rename y by v, or β by δ respectively in P, where v, δ are fresh (see Lemma 3.1.25). By activating the cuts, the intention is to force the renaming to take place first. We will also need to adopt our proposed strongly normalising rules (Section 3.1.3), which prevent a cut from deactivating, thereby enforcing priority to the renaming cuts.

Returning to the violation in the $(\textit{exp-imp}_{cbn})$ rule, in order to ensure the rewrite will be executed correctly (w.r.t. α-safety), we need to introduce an extra constraint to the applicability of the rule $(\textit{exp-imp}_{cbn})$, namely $\gamma \notin bs(\widehat{y}P\widehat{\beta}\cdot\alpha)$. (This can be equivalently formulated as $\gamma \notin bs(P) \wedge \beta \neq \gamma$). If the side-condition does not hold, then applying the rule will create a nested binding of (the image of) γ in the term graph.

To remedy the situation, renaming should take place. This implies that there are

now two alternatives for this rule:

$$(\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}P) \;\rightarrow\; (Q\widehat{\gamma}\dagger\widehat{y}R)\widehat{\beta}\dagger\widehat{z}P \qquad \twoheadleftarrow\ \gamma\neq\beta\notin bp(Q)$$
$$(\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}P) \;\rightarrow\; (Q\widehat{\gamma}\dagger\widehat{y}(R\widehat{\beta}\nleftarrow\widehat{v}\langle v\cdot\delta\rangle))\widehat{\delta}\dagger\widehat{z}P \twoheadleftarrow\ (\beta=\gamma\vee\beta\in bp(Q))$$

We will adopt a convention for naming modified rules. If the rule was originally called *rule*, then it is called:

- *rule*, if no renamings are involved in the rule.
- $rule^{rn\text{-}p}$, if a plug is renamed.
- $rule^{rn\text{-}s}$, if a socket is renamed.
- $rule^{rn\text{-}ps}$, if a plug *and* a socket is renamed.

Under this convention, the two variants of the $(exp\text{-}imp_{cbn})$ rule shown above are called $(exp\text{-}imp_{cbn})$ and $(exp\text{-}imp^{rn\text{-}p}_{cbn})$ respectively.

Likewise, the rules $(exp\text{-}imp_{cbv})$ and $(exp\text{-}imp^{rn\text{-}s}_{cbv})$ are defined respectively as:

$$(\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}P) \;\rightarrow\; Q\widehat{\gamma}\dagger\widehat{y}(R\widehat{\beta}\dagger\widehat{z}P) \qquad \twoheadleftarrow\ y\neq z, y\notin bs(P)$$
$$(\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}P) \;\rightarrow\; Q\widehat{\gamma}\dagger\widehat{v}((\langle v\cdot\delta\rangle\widehat{\delta}\nleftarrow\widehat{y}R)\widehat{\beta}\dagger\widehat{z}P) \twoheadleftarrow\ y=z\vee y\in bs(P)$$

Applying this solution to Example 4.2.1, we have, instead of the problematic step

$$(\widehat{y}\langle y\cdot\boldsymbol{\mu}\rangle\widehat{\boldsymbol{\mu}}\cdot\gamma)\widehat{\gamma}\dagger\widehat{k}((\widehat{y}\langle y\cdot\boldsymbol{\mu}\rangle\widehat{\boldsymbol{\mu}}\cdot\delta)\widehat{\delta}\,[k]\,\widehat{w}\langle w\cdot\alpha\rangle) \;\rightarrow\; (exp\text{-}imp_{cbn})$$
$$((\widehat{y}\langle y\cdot\boldsymbol{\mu}\rangle\widehat{\boldsymbol{\mu}}\cdot\delta)\widehat{\delta}\dagger\widehat{y}\langle y\cdot\boldsymbol{\mu}\rangle)\widehat{\boldsymbol{\mu}}\dagger\widehat{w}\langle w\cdot\alpha\rangle$$

the correction

$$\begin{aligned}
&(\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\gamma)\widehat{\gamma}\dagger\widehat{k}((\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\delta)\widehat{\delta}\,[k]\,\widehat{w}\langle w\cdot\alpha\rangle) &&\rightarrow\ (exp\text{-}imp_{cbn})\\
&((\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\delta)\widehat{\delta}\dagger\widehat{y}(\langle y\cdot\mu\rangle\widehat{\mu}\nleftarrow\widehat{v}\langle v\cdot\beta\rangle))\widehat{\beta}\dagger\widehat{w}\langle w\cdot\alpha\rangle &&\rightarrow\ (\nleftarrow cap)\\
&((\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\delta)\widehat{\delta}\dagger\widehat{y}\langle y\cdot\beta\rangle)\widehat{\beta}\dagger\widehat{w}\langle w\cdot\alpha\rangle &&\rightarrow\ (exp\text{-}rn)\\
&(\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\beta)\widehat{\beta}\dagger\widehat{w}\langle w\cdot\alpha\rangle &&\rightarrow\ (exp\text{-}rn)\\
&\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\alpha
\end{aligned}$$

To guarantee α-safety of the entire rewrite system, we need to make similar changes to each rule where a possible conflict is introduced. Take for example the rule that propagates an active cut over an inactive cut,

$$(\nleftarrow cut)\text{:}\quad P\widehat{\alpha}\nleftarrow\widehat{x}(Q\widehat{\beta}\dagger\widehat{y}R)\;\rightarrow\;(P\widehat{\alpha}\nleftarrow\widehat{x}Q)\widehat{\beta}\dagger\widehat{y}(P\widehat{\alpha}\nleftarrow\widehat{x}R)$$

There are two points of concern here: when $\alpha{=}\beta$, and if β or y occurs in $bc(P)$ (notice that $x{\neq}y$ as, by assumption, the left-hand side is an α-safe circuit). With this in mind, the rule $(\nleftarrow cut)$ is amended with extra side conditions and replaced

by the following variants (where v, δ are *fresh*):

$$
\begin{array}{rrcll}
(\backslash\!\!\backslash cut): & P\widehat{\alpha} \backslash\!\!\backslash \widehat{x}(Q\widehat{\beta} \dagger \widehat{y}R) & \rightarrow & (P\widehat{\alpha} \backslash\!\!\backslash \widehat{x}Q)\widehat{\beta} \dagger \widehat{y}(P\widehat{\alpha} \backslash\!\!\backslash \widehat{x}R) & \leftarrow C1 \\
(\backslash\!\!\backslash cut^{rn\text{-}p}): & P\widehat{\alpha} \backslash\!\!\backslash \widehat{x}(Q\widehat{\beta} \dagger \widehat{y}R) & \rightarrow & (P\widehat{\alpha} \backslash\!\!\backslash \widehat{x}Q)\widehat{\beta} \dagger \widehat{v}(P\widehat{\alpha} \backslash\!\!\backslash \widehat{x}(\langle v{\cdot}\delta\rangle\widehat{\delta} \backslash\!\!\backslash \widehat{y}R)) & \leftarrow C2 \\
(\backslash\!\!\backslash cut^{rn\text{-}s}): & P\widehat{\alpha} \backslash\!\!\backslash \widehat{x}(Q\widehat{\beta} \dagger \widehat{y}R) & \rightarrow & (P\widehat{\alpha} \backslash\!\!\backslash \widehat{x}(Q\widehat{\beta} \not\dagger \widehat{v}\langle v{\cdot}\delta\rangle))\widehat{\delta} \dagger \widehat{y}(P\widehat{\alpha} \backslash\!\!\backslash \widehat{x}R) & \leftarrow C3 \\
(\backslash\!\!\backslash cut^{rn\text{-}ps}): & P\widehat{\alpha} \backslash\!\!\backslash \widehat{x}(Q\widehat{\beta} \dagger \widehat{y}R) & \rightarrow & (P\widehat{\alpha} \backslash\!\!\backslash \widehat{x}(Q\widehat{\beta} \not\dagger \widehat{v}\langle v{\cdot}\delta\rangle))\widehat{\delta} \dagger \widehat{v}(P\widehat{\alpha} \backslash\!\!\backslash \widehat{x}(\langle v{\cdot}\delta\rangle\widehat{\delta} \backslash\!\!\backslash \widehat{y}R)) & \leftarrow C4
\end{array}
$$

and the side-conditions are:

$$
\begin{aligned}
C1 &= \beta \notin bp(P) \wedge \beta \neq \alpha \quad \wedge y \notin bs(P) \\
C2 &= (\beta \in bp(P) \vee \beta = \alpha) \wedge y \notin bs(P) \\
C3 &= \beta \notin bp(P) \wedge \beta \neq \alpha \quad \wedge y \in bs(P) \\
C4 &= (\beta \in bp(P) \vee \beta = \alpha) \wedge y \in bs(P)
\end{aligned}
$$

Almost all propagation rules (exceptions are $(\not\dagger flip)$, $(cap \not\dagger)$, $(flip \backslash\!\!\backslash)$, and $(\backslash\!\!\backslash cap)$) should be treated like this. Of the logical rules, only the two variants of the rule $(exp\text{-}imp)$ needs dealing with as specified above, giving a much more complicated rewriting system with a great many rewrite rules. The advantage of this approach is that name clash and capture are detected and dealt with, as stated by the following,

Theorem 4.2.7 *Let $P \rightarrow_\alpha Q$ stand for the notion of rewriting on $\mathcal{X}$ obtained by changing the rules as above. Then: if P is α-safe, and $P \rightarrow_\alpha Q$, then Q is α-safe.*

Proof 4.2.8 *Straightforward, by inspecting the rules.* □

The computational cost is low compared to the approach defined in Section 4.2.1 (see also Section 4.3); the price to pay is an increase in the number of rules. Since the detection of a possible α-safety violation in a rule is straightforward, it is even possible to, at the user level, allow for the definition of the normal rules, and to automatically generate the α-safe variants.

4.2.3 Avoiding Clash and Capture

Preserving Barendregt's convention is a perfectly adequate solution for maintaining the variable binding and variable identity relations: it forbids a term with nested binders to the same name to be created, and thereby totally avoids any ambiguity within the system. However, one can justifiably argue that the convention is restrictive and expensive to uphold at run-time. A more direct approach would

be to relax on Barendregt's convention, allowing names to occur both bound and free, assuming that the innermost binding binds strongest, and try and detect and avoid exactly the cases when name capture and name clash arise.

In the solution described in the previous section, free and bound connectors are all different, so capture is impossible. In the solution we propose here, connectors will be allowed to appear both free and bound. Instead the modification required on the rules is that, upon its application to a circuit, they should detect possible captures of and clashes between connectors. For example, referring back to Circuit (4.9) in Example 4.2.2, the application of the rule $(\mathit{exp\text{-}ins}\not\curlywedge)$ to the term

$$(((\widehat{y}\langle y\cdot\mu\rangle\widehat{\boldsymbol{\mu}}\cdot\gamma)\widehat{\delta}\not\curlywedge\widehat{y}\langle y\cdot\boldsymbol{\mu}\rangle)\widehat{\gamma}\dagger\widehat{x}\langle x\cdot\mu\rangle)\widehat{\mu}\dagger\widehat{w}\langle w\cdot\alpha\rangle$$

should first check if any of the binders of the export on the left occurs free in the capsule on the right. Specifically, it should ask if y or μ occur free in $\langle y\cdot\mu\rangle$; such a test would be positive, indicating, in this instance, the application of the rule would cause names to be captured.

We will show that we can always detect capturing safely, and perform the necessary α-conversion only then. The solution will, in appearance, be strikingly similar to that of Section 4.2.2 for the fact that *freeness* is used rather than *boundness*. In Section 4.3, this approach will be shown to be much more efficient; this is mainly because the solution of Section 4.2.2, many circuits which are not 'α-safe' (Definition 4.2.6) are left untouched here.

The original idea for the solution presented in this section comes from name clash and capture can be dealt with in the context of Bloo and Rose's calculus of explicit substitutions, $\lambda\mathbf{x}$, as was discussed in Section 2.4.2.

Let us consider the rule $(\mathit{exp\text{-}imp}_{cbn})$:

$$(\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}P)\;\rightarrow\;(Q\widehat{\gamma}\dagger\widehat{y}R)\widehat{\beta}\dagger\widehat{z}P\;\leftarrow\;\alpha, x\ \mathit{introduced}$$

In order to allow the rewrite to be executed like this, the side condition should express an extra criterion to avoid the capture of a free β in Q; if $\beta\in fs(Q)$, then the rule would bring that β under the binder $\widehat{y}$ on the right-hand side, and renaming should take place. Also, notice that if $\beta=\gamma$, there would be no capture, since the order of nested binders are preserved. This implies that there are now two alternatives for the rule $(\mathit{exp\text{-}imp}_{cbn})$. Where v,δ are fresh, we define $(\mathit{exp\text{-}imp}_{cbn})$

and $(\mathit{exp\text{-}imp}^{\mathit{rn\text{-}p}}_{\mathit{cbn}})$ respectively as:

$$\begin{array}{lll}
(\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}P) & \rightarrow (Q\widehat{\gamma}\dagger\widehat{y}R)\widehat{\beta}\dagger\widehat{z}P & \leftarrow\ \beta\notin fp(Q)\vee\beta=\gamma \\
(\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}P) & \rightarrow (Q\widehat{\gamma}\dagger\widehat{y}(R\widehat{\beta}\nleftarrow\widehat{v}\langle v\cdot\delta\rangle))\widehat{\delta}\dagger\widehat{z}P & \leftarrow\ \beta\in fp(Q)\wedge\beta\neq\gamma
\end{array}$$

Likewise, there the respective rules $(\mathit{exp\text{-}imp}_{\mathit{cbv}})$ and $(\mathit{exp\text{-}imp}^{\mathit{rn\text{-}s}}_{\mathit{cbv}})$ are, (where v,δ are fresh):

$$\begin{array}{lll}
(\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}P) & \rightarrow Q\widehat{\gamma}\dagger\widehat{y}(R\widehat{\beta}\dagger\widehat{z}P) & \leftarrow\ y\notin fs(P)\vee y=z \\
(\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}P) & \rightarrow Q\widehat{\gamma}\dagger\widehat{v}((\langle v\cdot\delta\rangle\widehat{\delta}\nleftarrow\widehat{y}R)\widehat{\beta}\dagger\widehat{z}P) & \leftarrow\ y\in fs(P)\wedge y\neq z
\end{array}$$

Also, since now we explicitly allow for connectors to occur both free and bound in a circuit, the rules need to check if the connector we try to connect to in a cut is actually really free. For example, rule $(\nleftarrow\mathit{cut})$ now becomes:

$$\begin{array}{rlll}
(\nleftarrow\mathit{cut}): & P\widehat{\alpha}\nleftarrow\widehat{x}(Q\widehat{\beta}\dagger\widehat{y}R) & \rightarrow (P\widehat{\alpha}\nleftarrow\widehat{x}Q)\widehat{\beta}\dagger\widehat{y}(P\widehat{\alpha}\nleftarrow\widehat{x}R) & \leftarrow C1 \\
(\nleftarrow\mathit{cut}^{\mathit{rn\text{-}p}}): & P\widehat{\alpha}\nleftarrow\widehat{x}(Q\widehat{\beta}\dagger\widehat{y}R) & \rightarrow (P\widehat{\alpha}\nleftarrow\widehat{x}Q)\widehat{\beta}\dagger\widehat{v}(P\widehat{\alpha}\nleftarrow\widehat{x}(\langle v\cdot\delta\rangle\widehat{\delta}\nleftarrow\widehat{y}R)) & \leftarrow C2 \\
(\nleftarrow\mathit{cut}^{\mathit{rn\text{-}s}}): & P\widehat{\alpha}\nleftarrow\widehat{x}(Q\widehat{\beta}\dagger\widehat{y}R) & \rightarrow (P\widehat{\alpha}\nleftarrow\widehat{x}(Q\widehat{\beta}\nleftarrow\widehat{v}\langle v\cdot\delta\rangle))\widehat{\delta}\dagger\widehat{y}(P\widehat{\alpha}\nleftarrow\widehat{x}R) & \leftarrow C3 \\
(\nleftarrow\mathit{cut}^{\mathit{rn\text{-}ps}}): & P\widehat{\alpha}\nleftarrow\widehat{x}(Q\widehat{\beta}\dagger\widehat{y}R) & \rightarrow (P\widehat{\alpha}\nleftarrow\widehat{x}(Q\widehat{\beta}\nleftarrow\widehat{v}\langle v\cdot\delta\rangle))\widehat{\delta}\dagger\widehat{v}(P\widehat{\alpha}\nleftarrow\widehat{x}(\langle v\cdot\delta\rangle\widehat{\delta}\nleftarrow\widehat{y}R)) & \leftarrow C4
\end{array}$$

and the side-conditions are,

$$\begin{array}{l}
C1 = y\notin fs(P)\wedge y\neq x\quad\wedge\ \beta\notin fp(P)\wedge\beta\neq\alpha \\
C2 = y\notin fs(P)\wedge y\neq x\quad\wedge\ (\beta\in fp(P)\vee\beta=\alpha) \\
C3 = (y\in fs(P)\vee y=x)\ \wedge\ \beta\notin fp(P)\wedge\beta\neq\alpha \\
C4 = (y\in fs(P)\vee y=x)\ \vee\ (\beta\in fp(P)\vee\beta=\alpha)
\end{array}$$

The reduction of Example 4.2.2, from Circuit (4.9), should have been:

$$\begin{array}{lll}
(4.9) = & (((\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\gamma)\widehat{\delta}\nleftarrow\widehat{y}\langle y\cdot\mu\rangle)\widehat{\gamma}\dagger\widehat{x}\langle x\cdot\mu\rangle)\widehat{\mu}\dagger\widehat{w}\langle w\cdot\alpha\rangle & \rightarrow (\mathit{exp\text{-}ins}\nleftarrow) \\
& ((\widehat{y}((\langle y\cdot\mu\rangle\widehat{\boldsymbol{\mu}}\nleftarrow\widehat{\boldsymbol{k}}\langle\boldsymbol{k}\cdot\boldsymbol{\tau}\rangle)\widehat{\delta}\nleftarrow\widehat{y}\langle y\cdot\mu\rangle)\widehat{\boldsymbol{\tau}}\cdot\gamma)\widehat{\gamma}\dagger\widehat{x}\langle x\cdot\mu\rangle)\widehat{\mu}\dagger\widehat{w}\langle w\cdot\alpha\rangle & \rightarrow (\nleftarrow d), (\mathit{cap\text{-}rn}) \\
& ((\widehat{y}(\langle y\cdot\tau\rangle\widehat{\delta}\nleftarrow\widehat{y}\langle y\cdot\boldsymbol{\mu}\rangle)\widehat{\boldsymbol{\tau}}\cdot\gamma)\widehat{\gamma}\dagger\widehat{x}\langle x\cdot\mu\rangle)\widehat{\mu}\dagger\widehat{w}\langle w\cdot\alpha\rangle &
\end{array}$$

As can be seen, the bound $\hat{\mu}$ of the export is forcefully renamed to $\hat{\tau}$, before the cut $\widehat{\delta}\dagger\widehat{y}$ can propagate through its structure. In the last step, we highlight that there is no conflict between the (previously captured) free μ of the capsule and the renamed binder of the export, τ.

All the rules need to be modified to check for possible capture of connectors. Although the structure of these new rules is similar to those in Section 4.2.2, the improvement in execution speed is impressive, as can be seen in the last section of this chapter.

4.3 Reduction Strategies for CTGRS

In Section 4.2 we studied different schemes for avoiding name clash and name capture in the context of the $\mathcal{X}$-calculus. We would like to directly compare the cost of upholding a particular safety criteria when used to reduce expressions.

To allow for a fair and accurate comparison across the schemes we proposed, it is important that, aside from the α-conversion steps, the same reduction paths are chosen when evaluating a term. This implies the need for a *deterministic* reduction strategy. Furthermore, this strategy should not be affected by any renaming cuts performing α-conversions. The following example shows how a naïve reduction strategy (in the call-by-name subsystem) could be affected, and motivates the need for an extension to the CTGRS implementation to allow for complex strategies to be defined.

Example 4.3.1 *Consider an instance of a graph where the rule exp-imp$^{rn\text{-}p}_{cbn}$ from the 'avoiding clash and capture' solution (Section 4.2.3) is applicable. Let us assume the subgraphs Q and R are pure and R introduces y. We have the following reduction.*

$$\begin{array}{ll} 1.\ (\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(\langle v\cdot\beta\rangle\widehat{\delta}\,[x]\,\widehat{z}Q) & \rightarrow (\textit{exp-imp}^{rn\text{-}p}_{cbn}) \\ 2.\ (\langle v\cdot\beta\rangle\widehat{\delta}\dagger\widehat{y}(R\widehat{\beta}\nprec\widehat{f}\langle f\cdot\sigma\rangle))\widehat{\sigma}\dagger\widehat{z}Q & \end{array}$$

If the α-conversion steps had been instantaneous, we would have obtained:

$$2'.\ (\langle v\cdot\beta\rangle\widehat{\delta}\dagger\widehat{y}R')\widehat{v}\dagger\widehat{z}Q \qquad \textit{where } R' = R\{\sigma/\beta\}$$

In the above circuit $(2')$, since R' introduces y, a renaming rule would be directly applicable. However, in step (2), although R still introduces y, we cannot evaluate the cut $\widehat{\delta}\dagger\widehat{y}$ without activating it first; this will cause it to (wastefully) propagate through R' (once the renaming cut $\widehat{\beta}\nprec\widehat{f}$ has been evaluated) searching for sockets named y—even though there is only one socket named y and it occurs at the topmost level. Each time this effect is observed, the cost of the α-conversion scheme will be skewed by a factor proportional to the size of the subgraph R when no optimization is used, and by a constant factor when the (optimized) garbage collection and renaming rules are used (Lemmas 3.1.21 and 3.1.25).

One way of avoiding the above problem is to ensure the reduction strategy prioritises reduction of the extra cuts which perform α-conversions. How these extra cuts are identified from renaming cuts that belong to the original circuit is also a point that needs to be addressed.

In Example 4.3.1, we would need to evaluate the renaming cut $\widehat{\beta} \nleftarrow \widehat{f}$ in step (2), before the other cuts in the expression. This action would rightly prevent the cut $\widehat{\delta} \dagger \widehat{y}$ activating, then propagating, to the right.

Although the $\mathcal{X}$-calculus specifies three kinds of cut (inactive, left-activated and right-activated), an input term will only contain instances of inactive cuts. Recall that the flagged cuts are internal operations that direct the propagation of a cut through the term structure.

In our system, we will adopt a convention where upon selection of a redex (i.e., a cut), the cut should run to *completion*. That is, when we choose to execute a redex $P\widehat{\alpha} \dagger \widehat{x}Q$, we connect *all* α's in P with *all* x's in Q.

To strengthen the motivation for our choice, the following example illustrates why a conventional outermost redex selection process is not a favourable strategy in the setting of the $\mathcal{X}$-calculus.

Example 4.3.2 (An Outermost Reduction Strategy in $\mathcal{X}$) *We begin with a nesting of cuts between* normal *circuits P, Q, R. In the call-by-value subsystem, a left-most outermost reduction strategy is applied to the graph root node, which traverses the graph structure attempting to match each rewrite rule with the current graph node. After a successful rewrite, the redex-searching process restarts from the root node of the term graph. Recall that active cuts cannot propagate over each other.*

1. $(P\widehat{\beta} \dagger \widehat{y}Q)\widehat{\delta} \dagger \widehat{z}R \quad \rightarrow (act\text{-L})$
2. $(P\widehat{\beta} \dagger \widehat{y}Q)\widehat{\delta} \nleftarrow \widehat{z}R \quad \rightarrow (cut\nleftarrow)$
3. $(P\widehat{\delta} \nleftarrow \widehat{z}R)\widehat{\beta} \dagger \widehat{y}(Q\widehat{\delta} \nleftarrow \widehat{z}R) \quad \rightarrow (act\text{-L})$
4. $(P\widehat{\delta} \nleftarrow \widehat{z}R)\widehat{\beta} \nleftarrow \widehat{y}(Q\widehat{\delta} \nleftarrow \widehat{z}R)$

In this relatively simple example, we are left in a situation resembling a traffic-jam. By step (4), the propagation of the outermost cut is blocked by the innermost active cuts. When the innermost cut propagates down the graph one level, the second innermost cut is permitted to propagate down one level. This pattern expands to more complicated examples, where each outer cut follows in the wake of an innermost cut (as would be seen in a traffic-jam, one car (cut) moves along a place, and each following car shifts along, filling the empty space).

The overall effect of this is an undesired increase in the cost of searching for the next redex (which involves graph traversal, structural matching and checking side-conditions).

During a graph rewrite step (Definition 2.4.25, new nodes may be added to the

graph. In step (1) of Example 4.3.2, an inactive cut is activated. Although the graph nodes Cut and CutL (for the cuts $\widehat{\delta} \dagger \widehat{z}$ and $\widehat{\delta} \not\vdash \widehat{z}$) are represented by two distinct node objects, our strategy must recognise that they are related in order to evaluate the cut to completion. To do this, we will define a strategy that upon selection of a redex, will sequentially apply a number of rules to the term graph to ensure that any activated cut is propagate through the structure of the term graph and evaluated to completion.

A generic language for specifying reduction strategies on term graphs has been proposed by Visser [87]. This language is rich enough to describe the kind of strategies we seek. The following section introduces the idea of *strategy combinators* and explains how to implement a reduction strategy for $\mathcal{X}$ that can reduce a cut to completion.

4.3.1 Strategy Combinators for CTGRS

Strategy combinators as defined by Visser [89, 87, 88] can be used to describe a complex traversal scheme for a term graph, during which the term graph can be modified, perhaps by applying a rewrite rule at the current node being visited. Examples of some strategies that may be specified in his language:

- "normalize the graph using a supplied list of rewrite rules, according to an innermost traversal"
- "repeatedly apply a rewrite rule to a node until failure"
- "visit all nodes at level three of the graph", and so on...

For our purposes, it is sufficient to restrict ourselves to a subset of the Visser's language, made up of the following combinators.

Definition 4.3.3 (Strategy Combinator Language, [87]) *The set of* strategy combinators *is defined by the set,*

$s ::= \mathsf{id}$	*Identity*
$\mid \mathsf{fail}$	*Fail*
$\mid \mathsf{L} \rightarrow \mathsf{R}$	*Rewrite rule*
$\mid [\mathsf{L} \rightarrow \mathsf{R}]$	*List of Rewrite rules*
$\mid \mathsf{seq}(\mathsf{s}, \mathsf{s})$	*Sequential Composition*
$\mid \mathsf{choice}(\mathsf{s}, \mathsf{s})$	*Left-biased choice*
$\mid \mathsf{all}(\mathsf{s})$	all *immediate successors*
$\mid \mathsf{one}(\mathsf{s})$	one *immediate successor*

Definition 4.3.4 (Application of a Strategy) *The application of a strategy to the term graph of a* CTGRS *is a pair consisting of a strategy combinator,* s*, and a rooted* CTGRS *subgraph* $(g|n)$*. The strategy combinator system has a global* fail flag *which, when raised, indicates a* fail *state; this fail state affects the operational behaviour of some of the combinators.*

We will write $\mathsf{s@}n$*, for the application of the strategy combinator* s *to the* CTGRS *graph rooted at* n*.*

The main behaviours result from the application of combinators are either, (i) the strategy results in another set of strategies being applied to some node(s) of the graph, (ii) the graph is modified by a rewrite rule, or (iii) the state of the global fail flag is altered.

Figure 4.1 accompanies the following description of strategy combinators.

- $\mathsf{id@}n$ is the identity strategy which simply leaves the supplied node unmodified.
- $\mathsf{fail@}n$ raises the fail flag indicating a state of failure.
- The application of a rewrite rule, $\mathsf{L{\rightarrow}R@}n$, (assuming the fail flag is not raised) attempts to match the rule head L with the subgraph rooted at n. If the match is successful, n will be rewritten to some subgraph rooted at n' as dictated by the rewrite rule; any further strategies to be applied to n are updated to refer to n'. If the match is unsuccessful, the fail flag is raised.
- The application of an ordered list of rewrite rules, $\mathsf{[L{\rightarrow}R]@}n$, (assuming the fail flag is not raised) sequentially traverses the list while attempting to apply each rewrite rule to n. The strategy terminates the traversal of the list upon the successful application of a rewrite rule. If the list is exhausted and no rule was applicable, the fail flag is raised.
- Assuming the fail flag is not raised, the application $\mathsf{seq(s1,s2)@}n$ sequentially applies its argument strategies, $\mathsf{s1}$ then $\mathsf{s2}$ to n. If either argument strategy raises the fail flag, the strategy aborts, leaving the system in a fail state.
- Assuming the fail flag is not raised, the application $\mathsf{choice(s1,s2)@}n$ attempts $\mathsf{s1@}n$, then performs $\mathsf{s2@}n$ if and only if $\mathsf{s1@}n$ raised the fail flag.

- Assuming the fail flag is not raised, $\mathsf{all}(\mathsf{s})@n$ attempts to apply s to each immediate successor (left-to-right) of the node n. Any successive applications are aborted if at any point the fail flag is raised by the application of s to the immediate successors of n.

- Assuming the fail flag is not raised, the application $\mathsf{one}(\mathsf{s})@n$ attempts to apply s left-to-right to a single immediate successor of n; if no successful application is found, the entire strategy fails.

In addition to these basic combinators, the language will allow user-definitions of more complex combinators. The *specification* for a user-defined combinator is given by the following construction (where here C is a variable over an infinite set of strings),

$$C(\mathsf{x}_1, \ldots, \mathsf{x}_k) = \mathsf{s}$$

The arguments $\mathsf{x}_1, \ldots, \mathsf{x}_n$ of C may occur, and are bound, in the definition body, s. The set of strategy combinators (Definition 4.3.3) is then extended with a user-defined combinator:

$$\begin{aligned} s ::= & \ldots \\ & \mid C(\mathsf{s}_1, \ldots, \mathsf{s}_k) \qquad \textit{User-defined combinator} \end{aligned}$$

An application of a user-defined combinator to a node n, $C(\mathsf{s}_1, \ldots, \mathsf{s}_k)@n$, denotes the instantiation $(\mathsf{s}\{\mathsf{s}_1/\mathsf{x}_1\}, \ldots, \{\mathsf{s}_k/\mathsf{x}_k\})$ of the body of s in the definition of C. Because this extension allows recursive strategies to be defined, we will dismiss nonsense definitions such as $C(\mathsf{x}) = C(\mathsf{x})$, by forbidding left-recursion.

We will also make use of some helper strategies.

Definition 4.3.5 (Helper Strategy Combinators, [89]) *We list below some user-defined helper combinators, followed by an informal description of the effect of applying the combinator to a node of some term graph. We assume before the application of each strategy, the fail flag is not raised.*

$\mathsf{try}(\mathsf{s}) = \mathsf{choice}(\mathsf{s}, \mathsf{id})$ *: attempt to apply the argument strategy* s *to the node. If* s *fails, clear the fail flag.*

$\mathsf{repeat}(\mathsf{s}) = \mathsf{try}(\mathsf{seq}(\mathsf{s}, \mathsf{repeat}(\mathsf{s})))$ *: repeatedly apply the strategy* s *to the node, until no more applications are possible, leaving the system in an unfail state.*

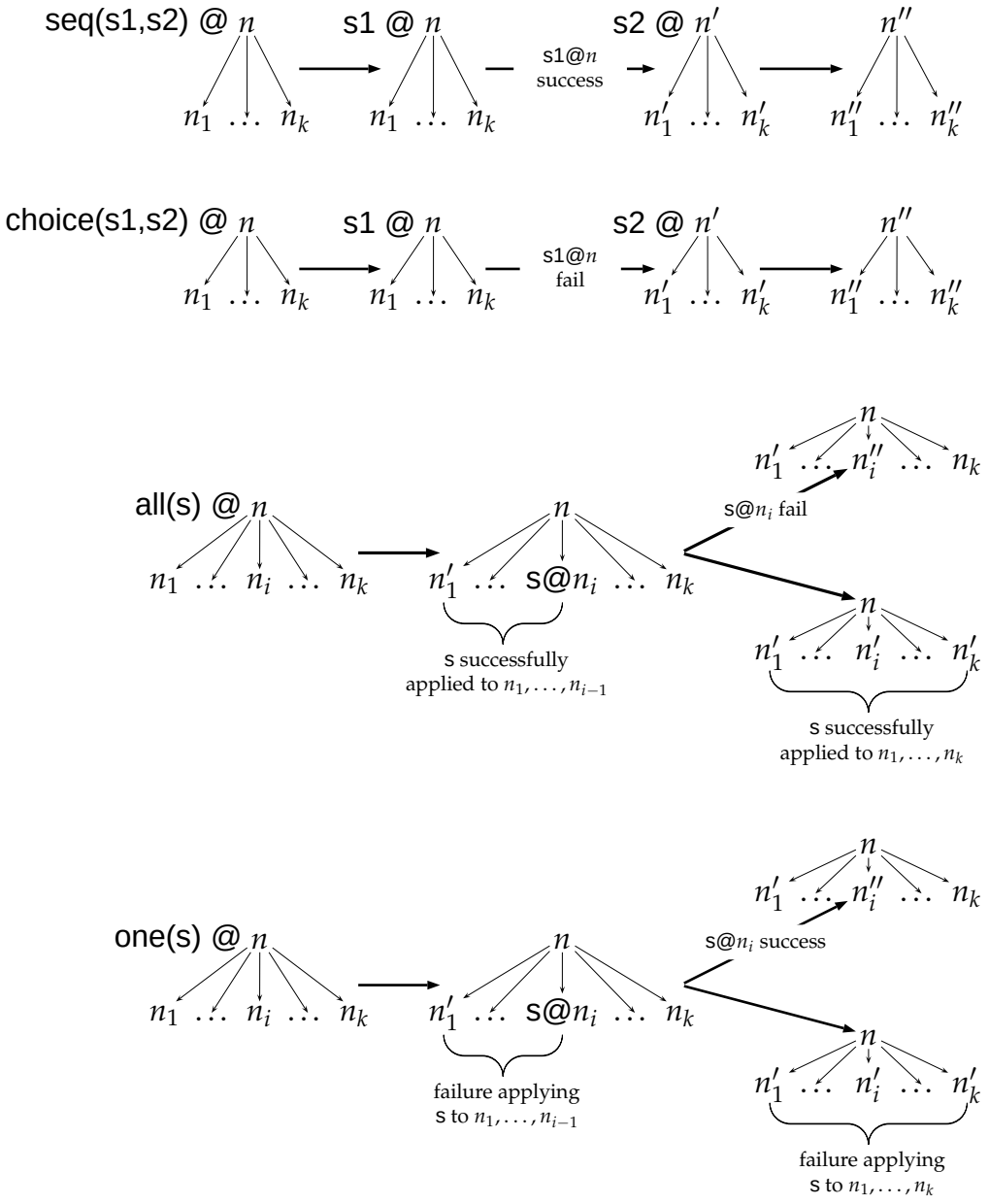


Figure 4.1: Applications of Basic Strategy Combinators to Arbitrary Graphs

$\mathsf{oncetd}(\mathsf{s}) = \mathsf{choice}(\mathsf{s}, \mathsf{one}(\mathsf{oncetd}(\mathsf{s})))$ *: search once top-down from the node and terminate after the first successful application of* s*. Raise the fail flag if no application was successful.*

$\mathsf{outermost}(\mathsf{s}) = \mathsf{repeat}(\mathsf{oncetd}(\mathsf{s}))$ *: search depth-first from the node and attempt to apply* s *to each node of the term graph; after a successful application restart the search from the node on which the strategy was first called (modulo rewriting of the subgraph rooted at that node).*

4.3.2 Reduction Strategies for $\mathcal{X}$

In this section we will define a strategy combinator that when be applied to $\mathcal{X}$-graphs will evaluate an inactive cut to completion. We will then extend this strategy to work with our proposed solutions to the problems of name clash and capture.

First, however, we will give a detailed example of the steps involved in applying a rewrite rule strategy to an $\mathcal{X}$-graph following a simple traversal scheme. The example is intended to mimic the steps taken by our implementation.

Example 4.3.6 (A reduction using oncetd**)** *In Figure 4.2, we illustrate the steps taken by our implementation of the strategy language to apply* $\mathsf{oncetd}(\mathit{cap\text{-}rn})$ *to the root of the term graph:*

$$g = \wr\langle y{\cdot}\gamma\rangle\widehat{\gamma}\,[z]\,\widehat{k}(\langle k{\cdot}\alpha\rangle\widehat{\alpha} \dagger \widehat{x}\langle x{\cdot}\mu\rangle)\int$$

During the application of a strategy, we maintain:

- a node stack*: a stack of nodes (in* g_X*) that records the path of the strategy through the term.*
- a combinator stack*: a stack of combinator states which records the progress of the strategy.*
- a fail flag*: to record whether a strategy has resulted in failure.*

The stack trace in Figure 4.2(a) begins with the root node of the term graph (Figure 4.2(c)) and the root node of the strategy graph (Figure 4.2(b)) on the node stack and combinator stack respectively; the fail flag is cleared. We write ch/*i to indicate the* choice *strategy is applying its* i^{th} *argument strategy, and* one/*j to indicate the* one *strategy is applying its argument strategy to the* j^{th} *successor of the node at which it was first applied. We summarize the interesting steps of the strategy below.*

1-2 *The zeroth argument of* choice *(the rule (cap-rn)) is pushed onto the combinator stack.*

3-4 *The failed match of the rule (cap-rn) with node 1, results in a failure state. The* choice *combinator recovers from the failure, expanding its second argument.*

5 *The* one *strategy pushes the zeroth successor of node 1 (i.e., node 2) onto the node stack, and pushes its argument strategy (*one*) onto the combinator stack. Notice that* one*'s argument strategy results in a recursive call being made to* choice.

6-9 *The application of the rule (cap-rn) to node 2 fails. The* choice *combinator recovers from the failure, pushing the* one *strategy onto the combinator stack. Since* one *is a traversal combinator, it pushes node 2's zeroth successor onto the node stack.*

10-14 *The application of the rule (cap-rn) to node 6 fails, and the* choice *strategy once again recovers from the failure. However, notice that this time the* one *strategy also fails, since node 6 has no successors.*

15 *The combinators are popped off the stack until a combinator is found that can reset the fail flag. The combinator happens to be the* one *combinator which was evaluating the zeroth successor of node 2.* one *clears the fail flag and proceeds to apply its argument strategy to the first successor of node 2 (i.e., node 7).*

16-19 *A repeat of the steps 10-14 occurs, except with node 7 on the node stack.*

20-21 *Upon returning to the* one *combinator which was visiting the first successor of node 2, it finds node 2 has no more successor. Therefore, the* one *strategy fails and propagates the failure state.*

22 *The* one *strategy failed to apply its argument strategy to the zeroth successor of node 1 (i.e., node 2). It recovers from the failure state and attempts to apply the same argument strategy to the first successor of node 1 (i.e. node 3).*

23-34 *This application fails, and one applies the argument strategy to the second successor of node 1 (i.e. node 4), then to the third successor of node 1 (node 5) when this fails.*

35 *The application of the rule (cap-rn) is successful and node 5 is rewritten to node* 8 *(see Figure 4.2(d)), and the node stack is updated.*

36-37 choice *does not evaluate its second argument strategy since the failure state is clear when it is the head of the combinator stack. The remaining combinators are popped off the stack and the strategy finishes.*

Our tool incorporates many optimisations that can bypass a significant number of stack operations. For example, noticing that the Cap nodes have only successor nodes which are names, steps 9-21 may be skipped.

At the end of Section 4.2.3, we remarked on some features we would require in an $\mathcal{X}$-calculus reduction strategy—these are summarised below.

Definition 4.3.7 (Criteria for Evaluating a Cut) *Given a pure $\mathcal{X}$-circuit, for any single* inactive *cut $P\widehat{\alpha} \dagger \widehat{x}Q$, a* good *$\mathcal{X}$-calculus reduction strategy will:*

1. *Evaluate the inactive cut to completion so that all circuits outputting on α in P are directly connected to all circuits in Q inputting from x, i.e., the resultant term should have eliminated the cut $\widehat{\alpha} \dagger \widehat{x}$ from the term.*
2. *Avoid mis-activating the cut in cases where alpha-conversion constructs block the direct application of logical rules to the cut.*

In the following we will formulate a 'good' reduction strategy for the CBN $\mathcal{X}$-calculus.

Observing the reduction rules of the pure $\mathcal{X}$-calculus (Definitions 3.1.4 and 3.1.7), we notice that the right hand sides of the rules introduce either (i) new active cuts, (ii) new inactive cuts or (iii) no new cuts.

We first group the reduction rules of the $\mathcal{X}$-calculus (as 'ordered list' strategy combinators) according to these features (see below).

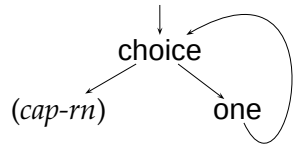
$$
\begin{aligned}
\mathsf{rename} &= [(cap), (exp\text{-}rn), (med\text{-}rn)] \\
\mathsf{logical} &= [(exp\text{-}imp_{cbn}), \mathsf{rename}]
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{activate} &= [(act\text{-}\mathrm{R}), (act\text{-}\mathrm{L})] \\
\mathsf{prop\_a1} &= [(exp\text{-}ins\not\leftarrow), (imp\not\leftarrow), (cut\not\leftarrow), (\nwarrow exp), (\nwarrow imp\text{-}ins), (\nwarrow cut)] \\
\mathsf{prop\_a2i0} &= [(exp\text{-}outs\not\leftarrow), (\nwarrow imp\text{-}outs)] \\
\mathsf{gc} &= [(cap\not\leftarrow), (\nwarrow cap)] \\
\mathsf{deact} &= [(\not\leftarrow d), (d \nwarrow)]
\end{aligned}
$$

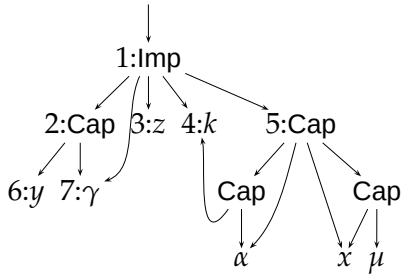
Using the above combinators, we will describe a user-defined strategy combinator for reducing an inactive cut so that the criteria outlined in Definition 4.3.7 is obeyed.

step	*Node Stack*	Combinator Stack	Fail
1	1	-	
2	1	ch/0, (*cap-rn*)	
3	1	ch/0	×
4	1	ch/1	
5	1,2	ch/1, one/0	
6	1,2	ch/1, one/0, ch/0, (*cap-rn*)	
7	1,2	ch/1, one/0, ch/0	×
8	1,2	ch/1, one/0, ch/1	
9	1,2,6	ch/1, one/0, ch/1, one/0	
10	1,2,6	ch/1, one/0, ch/1, one/0, ch/0, (*cap-rn*)	
11	1,2,6	ch/1, one/0, ch/1, one/0, ch/0	×
12	1,2,6	ch/1, one/0, ch/1, one/0, ch/1	
13	1,2,6	ch/1, one/0, ch/1, one/0, ch/1, one/0	
14	1,2,6	ch/1, one/0, ch/1, one/0, ch/1	×
15	1,2,7	ch/1, one/0, ch/1, one/1	
16	1,2,7	ch/1, one/0, ch/1, one/1, ch/0, (*cap-rn*)	
17	1,2,7	ch/1, one/0, ch/1, one/1, ch/0	×
18	1,2,7	ch/1, one/0, ch/1, one/1, ch/1	
19	1,2,7	ch/1, one/0, ch/1, one/1, ch/1, one/0	
20	1,2,7	ch/1, one/0, ch/1, one/1, ch/1	×
21	1,2,7	ch/1, one/0, ch/1, one/1	×
22	1,3	ch/1, one/1	
23	1,3	ch/1, one/1, ch/0, (*cap-rn*)	
24	1,3	ch/1, one/1, ch/0	×
25	1,3	ch/1, one/1, ch/1	
26	1,3	ch/1, one/1, ch/1, one/0	
27	1,3	ch/1, one/1, ch/1	×
28	1,4	ch/1, one/2	
29	1,4	ch/1, one/2, ch/0, (*cap-rn*)	
30	1,4	ch/1, one/2, ch/0	×
31	1,4	ch/1, one/2, ch/1	
32	1,4	ch/1, one/2, ch/1, one/0	
33	1,4	ch/1, one/2, ch/1	×
34	1,5	ch/1, one/3	
35	1,5	ch/1, one/3, ch/0, (*cap-rn*)	
36	1,8	ch/1, one/3, ch/0	
37	1	-	

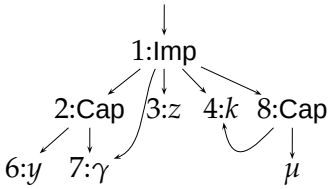
(a) Stack Traces



(b) oncetdstrategy



(c) steps 1–34



(d) steps 35–36

Figure 4.2: Application of oncetd(*cap-rn*) to $\langle y{\cdot}\gamma\rangle\widehat{\gamma}\,[z]\,\widehat{k}\,(\langle k{\cdot}\alpha\rangle\widehat{\alpha}\dagger\widehat{x}\langle x{\cdot}\mu\rangle)$

The first two combinators (rename and logical) completely reduce inactive cuts in one step. Their application is straightforward since they destroy the inactive cut being reduced. The next combinator (activate) is the only rule which turns an inactive cut into an active cut. If a cut is activated, we require that it will be propagated to completion. For now, we can assume a user defined combinator propagate() exists which carry out this task; clearly this strategy will make use of the remaining four strategy combinators, which deal only with inactive cuts. We can now define a strategy evalcut() which evaluates a cut to completion.

$$\mathsf{evalcut}() = \mathsf{choice}(\ \mathsf{logical},\ \mathsf{seq}(\ \mathsf{activate},\ \mathsf{propagate}()\ )\ ) \tag{4.11}$$

This strategy considers the *only* two cases of how to evaluate an inactivate cut: it can either be reduced by a logical rule, or activated then propagated through the term. The remainder of this section looks at how to define the propagate() combinator.

We can break down the work that needs to be done by the propagate() combinator into four cases:

1. propagate an active cut through a circuit that *does not* mention any connectors involved in the cut.
2. propagate an active cut through a circuit that *does* mention the connectors involved in the cut.
3. garbage collect the active cut, since it has reached the level of the capsules.
4. deactivate the cut, and attempt to reduce the deactivated cut.

The first two cases are covered by the following combinators.

$$\mathsf{seq}(\ \mathsf{prop\_a1},\ \mathsf{all}(\ \mathsf{try}(\ \mathsf{propagate}()\ )\ )\ ) \tag{4.12}$$

$$\mathsf{seq}(\ \mathsf{prop\_a2i0},\ \mathsf{seq}(\ \mathsf{all}(\ \mathsf{all}(\ \mathsf{try}(\ \mathsf{propagate}()\ )\ )\ ),\ \mathsf{evalcut}()\ )\ ) \tag{4.13}$$

These appear complicated, but are fairly straightforward to understand.

First we remark that by nesting several 'all' combinators, we can visit all the nodes at a particular depth of a term graph (relative to the node at which the strategy was first applied). For instance, $\mathsf{all}(\mathsf{s})@n$ will apply s to the nodes at a depth of 1 relative to the node n, while $\mathsf{all}(\mathsf{all}(\mathsf{s}))@n$ will apply s to the nodes of n at depth=2, and so on.

For (4.12), all the rules in the list prop_a1 have active cuts (that must be further propagated) at a depth of 1. Therefore, after a rule in prop_a1 has been applied,

the propagate() strategy is applied to the nodes at depth of 1 to propagate these newly introduced cuts.

For (4.13), all the rules of prop_a2i0 have active cuts at a depth of 2 *in addition to* an inactive at depth 0. The newly introduced active cuts at depth 2 must be further propagated, so there is a double nesting of the all combinator. The inactive cut at depth 0 also needs to be evaluated, so the evalcut() strategy is recursively applied; notice that to avoid the situation shown in Example 4.3.2, the active cuts are propagated *before* the inactive cut is evaluated.

The final two cases for the propagate strategy are described by the following combinators:

$$\mathsf{gc} \tag{4.14}$$

$$\mathsf{seq}(\ \mathsf{deact},\ \mathsf{evalcut}()\ ) \tag{4.15}$$

For (4.14), the strategy gc is applicable if the active cut is with a capsule that does not introduce the connector bound by the active cut. The rewrite rules state that in this case the active cut should be destroyed. Therefore, following a successful application of a gc strategy, no further work needs to be done: there is no more active cut to propagate.

For (4.15), if the strategy deact is applicable, the active cut is deactivated, creating a new inactive cut which must be evaluated; therefore the evalcut strategy is recursively applied to that inactive cut.

Combining these four parts, we obtain a definition for the propagate() strategy.

$$\begin{aligned}\mathsf{propagate}() = \ & \mathsf{choice}(\mathsf{gc}, \\ & \quad \mathsf{choice}(\ \mathsf{seq}(\ \mathsf{deact},\ \mathsf{evalcut}()\ ), \\ & \quad\ \mathsf{choice}(\ \mathsf{seq}(\ \mathsf{prop\_a1},\ \mathsf{all}(\ \mathsf{try}(\ \mathsf{propagate}()\ )\ )), \\ & \quad\quad \mathsf{seq}(\ \mathsf{prop\_a2i0},\ \mathsf{seq}(\ \mathsf{all}(\ \mathsf{all}(\ \mathsf{try}(\ \mathsf{propagate}()\ )\ )\ )\ ,\ \mathsf{evalcut}()\ )\ ), \\ & \quad)\ )\ )\end{aligned} \tag{4.16}$$

The nesting of 'choice' combinators ensures each propagation case is considered at the current node. If no case is successful, the strategy combinator leaves the system in the fail state.

The above discussion illustrates how to evaluate a cut to completion in the CBN $\mathcal{X}$-calculus. We would like to extend this strategy to the sets of rules that solve the name clash and name capture problems described in Section 4.2. This involves working with a larger set of rules, but the same idea of evaluating a cut exists.

We have two sets of rules to consider: the renaming using activated-cuts schemes and the lazy-copying α-conversion schemes with special rebinding symbols.

4.3.3 Alpha-conversion with Renaming Cuts

The strategy evalcut() is first extended to cater for the case of the $exp\text{-}imp^{rn\text{-}p}_{cbn}$ rule when a plug needs renaming. In this case, the active cut (at depth 2) is propagated through the term, i.e., $\mathsf{seq}(exp\text{-}imp^{rn\text{-}\mathsf{p}}_{\mathsf{cbn}}, \mathsf{all}(\,\mathsf{all}(\,\mathsf{try}(\,\mathsf{propagate}()\,)\,)\,)\,)$.

The propagate() strategy should then be modified to cater for any additional α-conversion structure. This involves partitioning this larger set of rules into lists of rules which have cuts (to be further propagated) at common depths. The renaming cuts, which will be the innermost active cuts, must then be given priority over the other cuts in the rule. We will consider the variant of the rule ($\nwarrow exp^{rn\text{-}s}$) shown below.

$$(\nwarrow exp^{rn\text{-}s}):\ Q\widehat{\alpha} \nwarrow \widehat{x}(\widehat{y}P\widehat{\beta}\cdot\delta) \ \rightarrow\ \widehat{w}(Q\widehat{\alpha} \nwarrow \widehat{x}(\langle w\cdot\mu\rangle\widehat{\mu} \nwarrow \widehat{y}P))\widehat{\beta}\cdot\delta$$

We omit the side-conditions of the rule, since they do not come into play in the following discussion.

This rule would have been placed in a 'list of rewrite rules' combinator named prop_a1a2, indicating the rule has active cuts at a depth of 1 and also at a depth of 2; in the rule above these are respectively the cuts $\widehat{\alpha} \dagger \widehat{x}$ and $\widehat{\mu} \dagger \widehat{y}$ on right-hand side of the rule. The propagation of active cuts for this combinator is described by the combinator:

$$\mathsf{seq}(\,\mathsf{prop\_a1a2},\,\mathsf{seq}(\,\mathsf{seq}(\,\mathsf{all}(\,\mathsf{all}(\,\mathsf{try}(\,\mathsf{propagate}()\,)\,)\,),\,\mathsf{all}(\,\mathsf{try}(\,\mathsf{propagate}()\,)\,)\,)\,)\,)$$

In other words, if a rule from the combinator prop_a1a2 is applied, propagate the inner active cuts at depth 2, and then propagate the remaining inner active cuts at depth 1. Lists of rules grouped together by the common depth of active and inactive cuts also need to be modified appropriately.

Now a 'good' outermost reduction strategy for the $\mathcal{X}$-calculus can be defined as:

$$\mathsf{outermost}(\,\mathsf{evalcut}()\,)$$

This strategy will search for the outermost inactive cut, which, when found, will apply the evalcut() strategy to that cut, evaluating that cut to completion.

4.3.4 Alpha-conversion with Rebinding Nodes

In this section, we look define a 'good' strategy for evaluating an inactive cut for the solution of lazy copying as defined in Section 4.2.1. We can follow a strategy similar to that of the previous section, except some extra care must be taken with regards to the rebinding nodes. The nodes which perform the rebinding (rs and rp) are not part of the $\mathcal{X}$-calculus and would ideally be transparent.

Barendsen and Smetsers remark in [16]: "mixing copy rules with the reduction rules in [the set of rewrite rules] may destroy properties of [the set of rewrite rules] such as confluence, or at least make it very difficult to check whether known properties of [the TGRS] extend to [the TGRS with copy rules added]". This turns out to be true in our case; the rebinding nodes interfere with structural matching and the 'introduces' side conditions of the rewrite rules. For example, the term $\mathsf{rs}(\langle x{\cdot}\alpha\rangle, \alpha, \beta)$ does not introduce β, although it evaluates to $\langle x{\cdot}\beta\rangle$.[2] A naïve solution could force the copying operation to completion, though this may turn out to be unnecessary.

We propose a lazier solution that *hides* the existence of rebinding nodes from the structural matching step of the rewriting procedure. Observing the lazy copying rewrite rules (Definition 4.2.4), this can be achieved by ensuring all rebinding nodes are at least two levels from any redex; in other words, a rebinding node should never be an immediate successor of a cut.

We specify a strategy pushRebind(), which when applied to a rebinding node, pushes that rebinding node down through the term-graph by one level. There are two cases to consider—either a rebinding rule (from Definition 4.2.3) is directly applicable to the current rebinding node, or not—i.e., there is a chain of one or more successive rebinding nodes prohibiting propagation. When such a chain exists, the strategy traverses to the lowest rebind node of that chain where a rule *will* be applicable. The lowest node is propagated further one level, followed in turn by each blocked ancestor.

The skeleton definition of this strategy is given below and makes use a group of all the rebinding evaluation rules (rebind_rules) as given in Definition 4.2.3, and a strategy repeat'(s) which repeatedly applies its argument strategy to the current

[2] We could alter our definition of *introduces* so that $\mathsf{rs}(\langle x{\cdot}\alpha\rangle, \alpha, \beta)$ introduces β, but this will still not side-step the problem of the rebinding node structure interfering with the graph matching process.

node or fails.

$$\mathsf{repeat}'() = \mathsf{seq}(\,\mathsf{s},\,\mathsf{repeat}'(\mathsf{s})\,) \tag{4.17}$$

$$\begin{aligned}\mathsf{pushRebind}() = {} & \mathsf{choice}(\,\mathsf{repeat}'(\,\mathsf{rebind\_rules}),\\ & \mathsf{choice}(\,\mathsf{seq}(\,\mathsf{all}(\,\mathsf{try}(\,\mathsf{pushRebind}()\,)\,),\,\mathsf{rebind\_rules}),\\ & \mathsf{fail})\,);\end{aligned} \tag{4.18}$$

The propagate() strategy can now be extended to make use of pushRebind(). For each rebinding node introduced by a rewrite rule of the $\mathcal{X}$-calculus, the pushRebind() strategy is applied to that node, guaranteeing it is never the successor of a cut.

4.3.5 Optimisations

We highlight a simple optimisation to the outermost strategy that will greatly decrease the search time for the next redex. Currently, after a successful reduction of an inactive cut, the search for the next inactive cut restarts from the point at which the original call to the strategy was made, i.e., the root node of the term graph. As a general outermost strategy, this is a safe course of action to take since an outermost redex may have been skipped while an inner redex is evaluated c.f. call-by-value reduction in the lambda-calculus. The $\mathcal{X}$-terms we evaluate are pure terms, and according to our evalcut() reduction strategy inactive cuts are evaluated to completion. Since inactive no cut can block propagating active cuts, the depth of the subsequent outermost cuts are therefore guaranteed to be at a depth lower or equal to the current node. Using this observation, we can define an outermost reduction strategy to continue redex-searching from the current node pointed to.

$$\mathsf{outermost}'(\mathsf{s}) = \mathsf{seq}(\,\mathsf{repeat}(\mathsf{s}),\,\mathsf{all}(\,\mathsf{try}(\,\mathsf{outermost}'(\mathsf{s})\,)\,)\,)$$

By parametising evalcut with an ordered list of activation rules and a variant of the (*exp-imp*) rule, we can define two outermost strategies for call-by-name and call-by-value as follows.

$$
\begin{aligned}
\mathsf{cbnact} &= [(act\text{-}\mathrm{R}), (act\text{-}\mathrm{L})] \\
\mathsf{cbvact} &= [(act\text{-}\mathrm{L}), (act\text{-}\mathrm{R})]
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{outermostCBN} &= \mathsf{outermost'}(\,\mathsf{evalcut}(\,\mathsf{cbnact},\, (exp\text{-}imp_{\mathsf{cbn}}))\,) \\
\mathsf{outermostCBV} &= \mathsf{outermost'}(\,\mathsf{evalcut}(\,\mathsf{cbvact},\, (exp\text{-}imp_{\mathsf{cbv}}))\,)
\end{aligned}
$$

4.3.6 Benchmarks

In this section we present our benchmarks comparing the costs of the solutions to name clash and capture we proposed in Section 4.2. In the previous section we described how to extended our CTGRS implementation with 'strategy combinators' in order to define a fair reduction strategy that could be used to compare the proposed solutions. Incorporating the strategy combinator language into our tool presented us with some problems, which we summarise below.

Visser has provided the community with a Java implementation of the strategy combinator framework, called JJTraveler. The framework allows modular extensions to the combinator language, allowing one to add *user-defined combinators* to the system by inheritance. A full description of this framework can be found in [89, 35].

Integration of the framework to our CTGRS implementation, also written in Java, was straightforward. Unfortunately, preliminary testing revealed the implementation was unable to traverse some of the larger term-graphs generated by our benchmarks (which can contain in excess of 300,000 nodes) resulting in stack-overflows. The reason for this was the heavy reliance on recursion due to the use of a modified Visitor design-pattern[3]. We chose to re-implement the framework taking an iterative approach instead. The set of strategy combinators (Definition 4.3.3) extended with 'user-defined combinators' allows recursive strategies to be built. These were implemented as cyclic graphs (following the implementation of JJTraveller).

To maintain the state of the strategy (i.e., to track how much of the strategy had been processed during an application), we used two stacks. The working details of these structures were exemplified in Example 4.3.6. Recall that the 'combinator stack' tracked the current position within the strategy combinator graph, and

[3]Simply increasing the stack size of the JVM was not seen to be a scalable solution. Although recent Java implementations do include recursion optimisations, these mainly work on performance. Since our benchmarks will count atomic operations rather than measure time, our main concern is heap usage, which we can manage more efficiently with an iterative approach.

the 'node stack' tracked the node of term graph which the current combinator was being applied to. The approach allowed us to obtain an accurate measure of the cost of *traversing* the graph searching for redexes: it was a count of the number of (constant-time) stack operations, plus the number of node matchings, plus the cost of checking the side-conditions. We were surprised to find that the search cost was often ignored in rewriting literature, with some measure of the cost of 'manipulating' the graph usually being quoted instead. Our preliminary results revealed that when large term-graphs were being traversed, the search time played a significant role in the cost of reducing the graph.

We chose to measure the running of (interpreted) λ-terms since these are well-known benchmarks [4, 65], and the efficiency of the various formalisms and abstract machines can be better compared. We can of course not confront their (published) run-time measurements because of differences in platforms and processor architectures.

We use the usual encoding for Church Numerals ($\mathsf{n} = \lambda xy.x^n y$). In addition, we use the combinators, $Q = (\lambda z.(\lambda x.zxxx)(\lambda y.\mathsf{2}(\lambda x.y(x\mathsf{I}))n))\mathsf{II}$, with n replaced with a chosen Church Numeral and $\mathsf{I} = \lambda x.x$.

Note that we do not wish consider the cost of encoding -terms to $\mathcal{X}$ as we are interested in comparing the efficiency of the α-conversion mechanisms for our proposed solutions. Because of this, we use Prawitz's normal form preserving encoding from natural deduction proofs to sequent calculus proofs as given in Definition 3.1.12.

Our benchmarking results are listed in Tables 4.2 and 4.3. For each test case we record the following two measurements:

Search Cost: : a count of atomic operations involved in traversing the graph and searching for redexes, i.e. the number of push/pop stack operations to evaluate the strategy, plus the number of attempted matchings made between the rule heads and graph nodes, plus the cost of testing the side-condition.

Rewrite Cost: : a count of the (more expensive) graph transforming operations, i.e. the number of nodes added and deleted plus number of edges added and deleted.

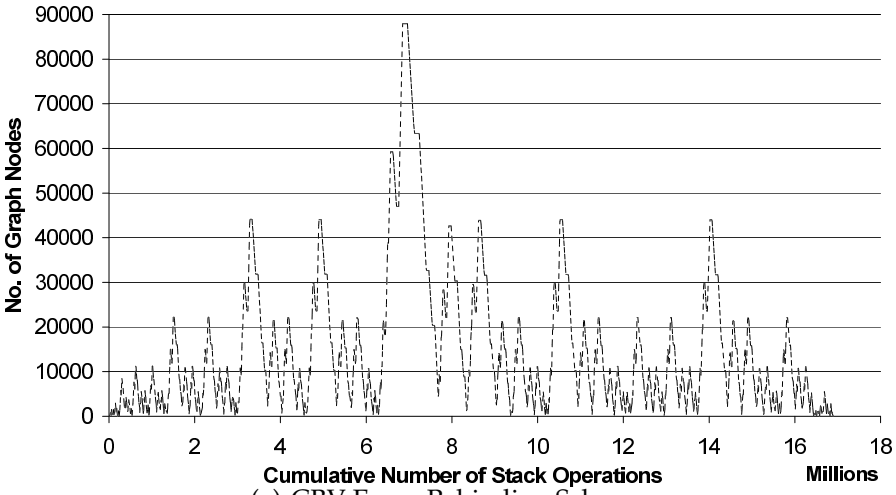
A straightforward numerical comparison of costs suggests the following relationship of efficiency between α-conversion schemes, under either reduction strategy (CBN or CBV).

TestCase	Rebinding-GC		Barendregt-NoGC		Barendregt-GC		Avoid Capture-GC	
	Search	Rewrite	Search	Rewrite	Search	Rewrite	Search	Rewrite
22II	0.861	0.163	0.264	0.0765	0.159	0.0421	0.0994	0.0330
222II	62.0	15.7	3.64	0.942	0.841	0.208	0.495	0.153
2222II	-	-	-	-	4650	715	1690	522
210II	0.372	0.0682	0.140	0.0401	0.0762	0.0214	0.0548	0.0194
2210II	2.40	0.471	0.649	0.165	0.269	0.0660	0.173	0.0541
22210II	225	56.7	10.6	1.84	1.53	0.305	0.805	0.236
$P2$	32.9	7.85	4.01	0.857	0.999	0.218	0.429	0.145
$P3$	51.4	12.5	5.54	1.11	1.20	0.256	0.487	0.165
$P5$	110	27.2	9.94	1.74	1.63	0.331	0.602	0.206
$P10$	429	109	31.4	4.01	2.98	0.519	0.890	0.309
$P20$	2240	575	143	11.6	6.71	0.895	1.47	0.514
$P50$	26100	6780	1550	58.4	26.3	2.02	3.19	1.13

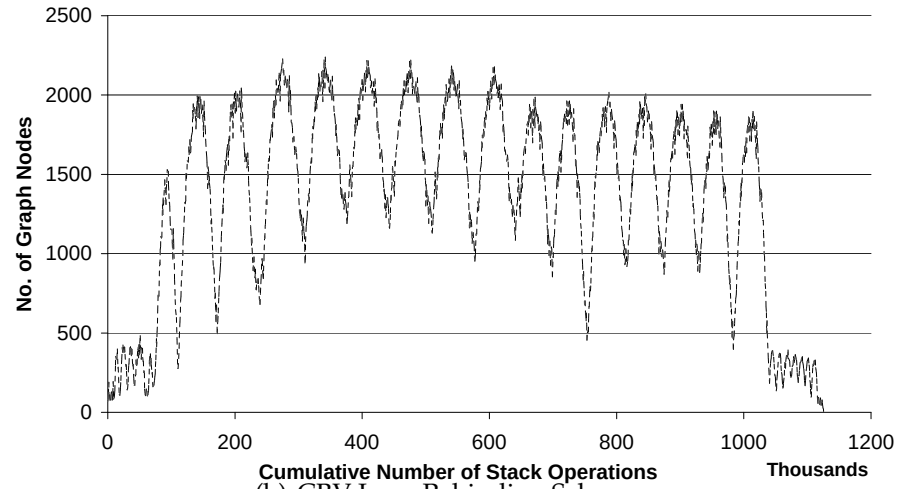
Table 4.2: CBV Results: Cost measured in units of 10^6 operations (to 3.s.f)

TestCase	Rebinding-GC		Barendregt-NoGC		Barendregt-GC		Avoid Capture-GC	
	Search	Rewrite	Search	Rewrite	Search	Rewrite	Search	Rewrite
22II	0.914	0.156	0.314	0.0908	0.173	0.0439	0.0963	0.0322
222II	17.0	3.27	3.86	1.01	1.42	0.305	0.551	0.178
2222II	-	-	146000	29700	13100	1460	2490	805
210II	0.458	0.0819	0.164	0.0441	0.0868	0.0227	0.0559	0.0198
2210II	2.12	0.360	0.726	0.177	0.315	0.0742	0.151	0.0507
22210II	32.1	6.38	7.29	1.60	2.11	0.441	0.754	0.246
$P2$	540	138	22.5	3.02	3.51	0.582	0.902	0.308
$P3$	875	224	30.6	3.67	4.23	0.670	1.04	0.354
$P5$	2000	515	52.6	5.22	5.85	0.847	1.32	0.447
$P10$	8670	2240	152	10.5	11.0	1.29	2.02	0.679
$P20$	49600	12900	624	27.2	25.9	2.17	3.41	1.14
$P50$	626000	163000	6050	126	107	4.81	7.59	2.54

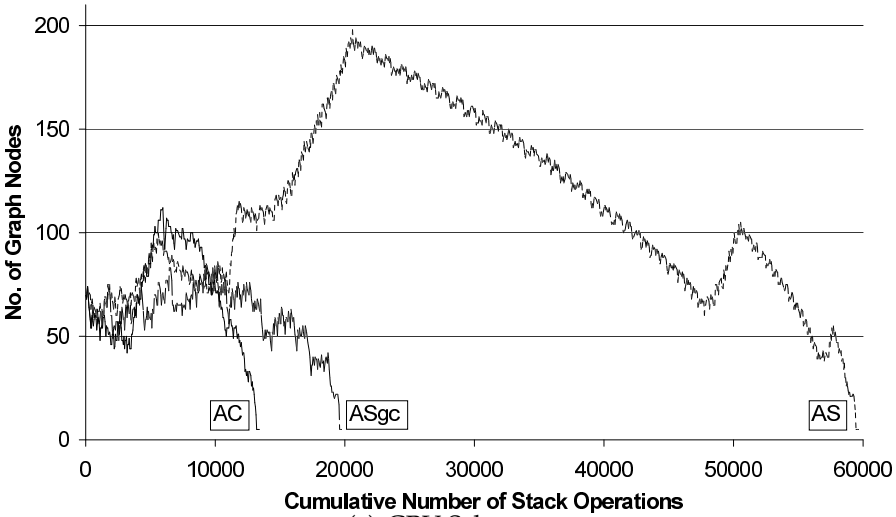
Table 4.3: CBN Results: Cost measured in units of 10^6 operations (to 3.s.f)



(a) CBV Eager Rebinding Scheme

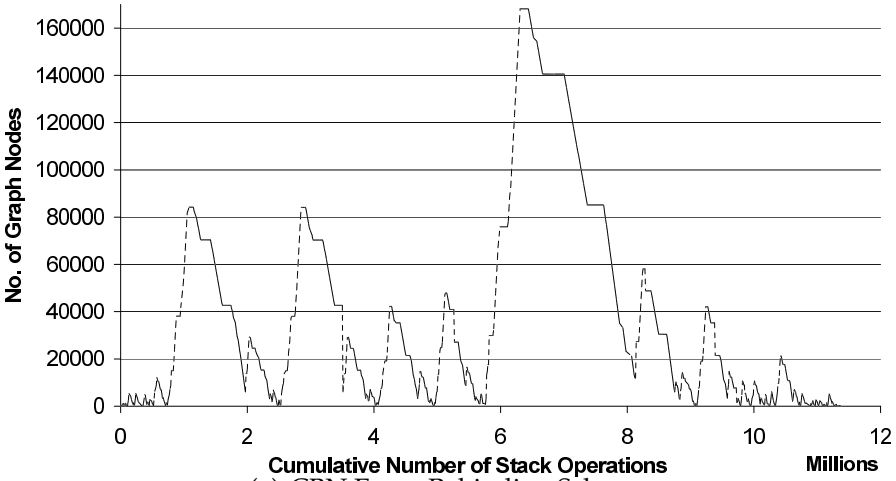


(b) CBV Lazy Rebinding Scheme

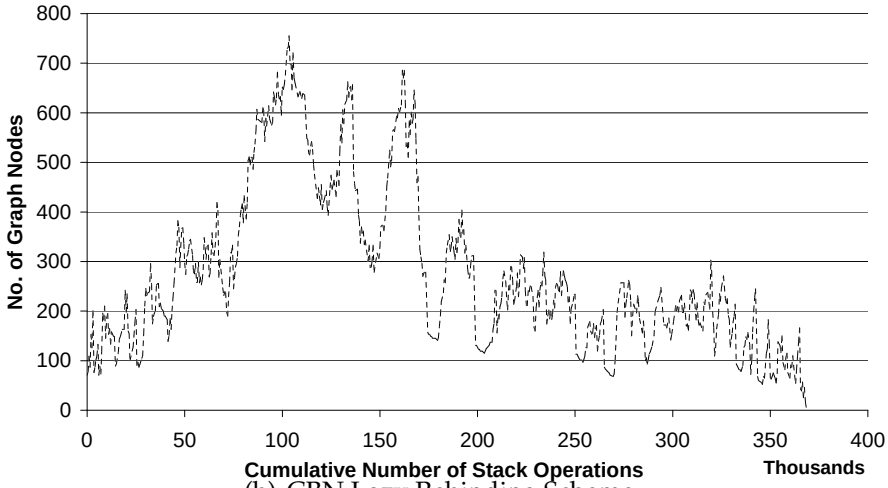


(c) CBV Schemes

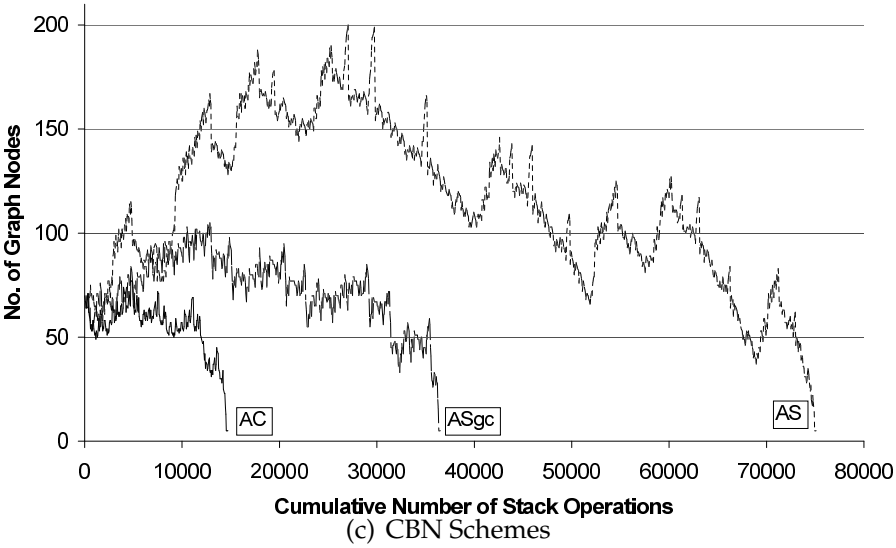
Figure 4.3: Variation of Graph Size over 'time' for reducing $\lceil\!\lceil 222II \rfloor\!\rfloor_\delta$ under different α-conversion schemes (AS=Barendregt-NoGC, ASgc=Barendregt-GC, AC=AvoidCapture-GC)



(a) CBN Eager Rebinding Scheme



(b) CBN Lazy Rebinding Scheme



(c) CBN Schemes

Figure 4.4: Variation of Graph Size over 'time' for reducing $\lceil\!\lceil 222II \rceil\!\rceil_\delta$ under different α-conversion schemes (AS=Barendregt-NoGC, ASgc=Barendregt-GC, AC=AvoidCapture-GC)

Rebinding GC $<$ Barendregt NOGC $<$ Barendregt GC $<$ Avoid Capture GC

Another observable trend is the linear relationship between the redex search cost and graph rewrite cost. As the size of the graph increases, the search cost increases since more operations are required to traverse the graph structure. Once a suitable redex (a cut) is found, the cost of reducing that cut is related to the size of its subterms since it must be propagated through them.

There is a significant difference between the cost of reduction under the rebinding scheme versus the other schemes tested. In fact, our original attempt at benchmarking the rebinding solution was not lazy at all, and often resulted in memory requirements greater than the 2GiB limit. We recorded the size of the graph (number of node objects) as the reduction progressed for each test-case in order to gain insight into this vast requirement of system resources. Results from the eager strategy are shown in Figures 4.4(a) and 4.3(a). As explained in Section 4.2.1, the rebinding scheme works by destroying the sharing in portions of the term-graph as so to guarantee that no binder of any redex is shared. This copying-out effect, seen as peaks in the graphs, shows an increase in the number of nodes whenever a cut is propagated through varying sizes of subterm causing all sharing to be destroyed. Looking closer at these graphs the cost of searching is also clearly visible. As the number of nodes in a graph increases, small horizontal 'platforms' can be observed. These regions represent pure search costs consisting only of traversal stack operations plus unsuccessful rule matches.

Switching to a lazy mechanism as detailed in Section 4.3.4, although still relatively expensive, kept the size of the graph low enough for many previously failed tests to run to completion. The results of the lazy strategy are shown in Figures 4.4(b) and 4.3(b). The CBV graph highlights nicely the copying out of each argument when it is supplied to a function.

We also investigated the variation of graph size under the other α-conversion schemes; the results are displayed in Figures 4.4(c) and 4.3(c). The shapes of these graphs appear to be less random than those from the rebinding scheme, and we see that for a particular reduction strategy (CBN or CBV) the overall shape of the graphs are similar. This is an expected side effect of the reduction strategy $\mathsf{evalcut}()$, designed to make α-conversions transparent from the point of view of the term being reduced. The reductions therefore only diverges at points where the need for α-conversions differ.

4.4 Chapter Summary

In this chapter we presented details of our implementation of the $\mathcal{X}$-calculus. We began by defining a conditional higher-order term graph rewrite system (CTGRS) which was expressive enough to express the key features of the $\mathcal{X}$-calculus, namely the binding relations and the side-conditions on rewrite rules.

Following this, we highlighted the problems of name clash and name capture in the $\mathcal{X}$-calculus, for which we proposed three different solutions. To compare the cost of these systems in a fair way, we extend our CTGRS with Visser's generic language for describing reduction strategies. Using this extension we described a complex strategy for the pure $\mathcal{X}$-calculus that is able to reduce a cut to 'completion'. We extended this to the three proposed systems that prevent name capture and name clash, allowing us to directly compare the cost of reductions.

In the final section, we presented some optimisations to our strategy followed by a quantitative evaluation of the three systems. As expected, the avoiding capture solution was the least expensive.

Chapter 5

Extending the $\mathcal{X}$-Calculus

In this chapter we study the relationship between the type system of the $\mathcal{X}$-calculus and Urban's variant of Kleene's G3a sequent calculus [86]. We will generalise the work of van Bakel *et al.* [9] and detail a generic method for building 'Curry-Howard' pairs of calculi (a term calculus whose type system corresponds to a logical calculus) in the style of $\mathcal{X}$.

It is common for computational calculi to be based on logics built from implication, since the computational behaviour associated with this connective is well understood. We will study some simple relationships between different (binary) logical connectives and computational calculi built from logics employing them as primitives, with the aim of determining the 'computational content' associated to the connective. In particular, we will investigate the simulation capabilities of calculi built from 'functionally complete' sets of connectives.

Noticing that, to our knowledge, some binary connectives (if-and-only-if and exclusive-or) have never been studied in a Curry-Howard setting, we will construct a term calculus, $\mathcal{X}^{\leftrightarrow}$, based on the if-and-only-if connective. We will study its computational properties and show it has unexpected simulation properties.

5.1 Proof Inhabitation and Types for Circuits

The goal of Urban's Ph.D is to develop a strongly normalising cut-elimination procedure for sequent calculus proofs of classical logic [86]. He chooses to work with, and formalises, Kleene's G3a sequent calculus[1] so that he can present his

[1] Kleene only mentions the calculus without formalising it, [58, pp. 481]

cut-elimination procedures in a "convenient form". Kleene's G3a sequent calculus features implicit rules for contraction and weakening rules, reducing the number of inference rules Urban needs to work with.

In devising this 'convenient form', Urban seeks a linear representation or *term annotation* for proof trees. Following conventions in type theory, he treats contexts as sets of $(label, formula)$ pairs, and "*not* as multisets". He introduces two classes of labels which he calls 'names' and 'co-names'; these couple respectively with formulas on the left and on the right of the turnstile of a sequent. In calculi built in the style of $\mathcal{X}$, the class of labels corresponds to the class of connectors of which there are two kinds: sockets and plugs respectively. A context is constrained so that it cannot contain more than one occurrence of a particular label; Urban calls this the *context convention*. We will refer to the G3a calculus with the above modifications as the G3a' sequent calculus.

Urban argues that λ-term annotations (along with some other existing proposals) could not capture the full structure of sequent calculus proofs, and would therefore lead to an incomplete cut-elimination procedure. Instead, he devises a more direct set of proof inhabitants that hold an exact correspondence with the proofs of G3a'; his set of 'raw terms' are defined as follows.

Definition 5.1.1 (Urban's Raw Terms [86]) *Let B and C be types, x, y, z names and a, b, c co-names; the (implicative) set of raw terms is defined by the following grammar.*

$$
\begin{array}{rll}
M, N ::= & \mathsf{Ax}(x, a) & \text{axiom} \\
\mid & \mathsf{Cut}(\langle a{:}B\rangle M, (x{:}B)N) & \text{cut} \\
\mid & \mathsf{Imp}_R((x{:}B)\langle a{:}C\rangle M, b) & \text{implication-right} \\
\mid & \mathsf{Imp}_L(\langle a{:}B\rangle M, (x{:}C)N, y) & \text{implication-left}
\end{array}
$$

Bound labels are identified using juxtaposition and brackets, i.e., $(x{:}A)M$ indicates $x{:}A$ is bound in M, and $\langle a{:}A\rangle M$ indicates $a{:}A$ is bound in M.

Notice that raw terms carry type information at the level of the syntax. Also notice that if these types were erased, and the syntax revised appropriately, one could obtain the syntax of the $\mathcal{X}$-calculus.

Inhabitation deals with assigning terms to proofs. The process opposite to inhabitation attempts to relate logical formulas to terms and builds proofs from programs. In the context of untyped calculi, such as the $\mathcal{X}$-calculus, the question of type assignment becomes relevant, as discussed in Section 2.3.1. (Type

assignment for a typed calculus, such as Urban's, is trivial). Given an $\mathcal{X}$-circuit, the question of whether a term is typeable is answered by the construction of (or failure to construct) a typing derivation according to the following formulations.

Definition 5.1.2 (Types and Contexts for $\mathcal{X}$ [9])

1. *The set of types ranged over by A, B, is defined over a set of* type-variables *$\{\varphi, \varphi_1, \varphi_2, \varphi_3, \ldots\}$ by the grammar:*

$$A, B ::= \varphi \mid A \rightarrow B$$

2. *A* context of sockets Γ *is a mapping from sockets to types, denoted as a finite set of* statements *$x{:}A$, such that the* subjects *of the statements (the sockets) are distinct. We write $\Gamma, x{:}A$ for the context defined by:*

$$\begin{aligned} \Gamma, x{:}A &= \Gamma \cup \{x{:}A\}, \text{ if } \Gamma \text{ is not defined on } x \\ &= \Gamma, \qquad\qquad \text{if } x{:}A \in \Gamma \end{aligned}$$

   Therefore, when writing a context as $\Gamma, x{:}A$, this implies that $x{:}A \in \Gamma$, or Γ is not defined on x. We write $\Gamma \backslash x$ for the context from which the statement concerning x, if any, has been removed.

3. *A* contexts of plugs Δ, *and the notations $\alpha{:}A, \Delta$ and $\Delta \backslash \alpha$ are defined in a similar way.*

Definition 5.1.3 (Typing for $\mathcal{X}$ [9])

1. Type judgements *are expressed via the ternary relation $P :\cdot \Gamma \vdash \Delta$, where Γ is a context of sockets, Δ is a context of plugs, and P is an $\mathcal{X}$-circuit. We say that P is the* witness *of this judgement.*

2. Type assignment for $\mathcal{X}$ *is defined by the following sequent calculus:*

$$\frac{}{\langle x{\cdot}\alpha \rangle :\cdot x{:}A, \Gamma \vdash \Delta, \alpha{:}A}\,(Ax) \qquad \frac{P :\cdot \Gamma \vdash \Delta, \alpha{:}A \quad Q :\cdot x{:}A, \Gamma \vdash \Delta}{P\widehat{\alpha} \dagger \widehat{x}Q :\cdot \Gamma \vdash \Delta}\,(Cut)$$

$$\frac{P :\cdot x{:}A, \Gamma \vdash \Delta, \beta{:}B}{\widehat{x}P\widehat{\beta}{\cdot}\alpha :\cdot \Gamma \vdash \Delta, \alpha{:}A \rightarrow B}\,(\rightarrow R) \qquad \frac{P :\cdot \Gamma \vdash \Delta, \alpha{:}A \quad Q :\cdot x{:}B, \Gamma \vdash \Delta}{P\widehat{\alpha}\,[y]\,\widehat{x}Q :\cdot y{:}A \rightarrow B, \Gamma \vdash \Delta}\,(\rightarrow L)$$

We write $P :\cdot \Gamma \vdash \Delta$ if there exists a derivation that has this judgement in the bottom line.

Notice that by erasing the witnesses and connectors from the type judgements, one obtains a formal system of logic.

Also, by the special meaning we associate to the comma in contexts, contraction is implicit. Since the axiom may contain arbitrary contexts Γ and Δ, weakening is also implicit.

As mentioned above in $P :\cdot \Gamma \vdash \Delta$, the circuit P acts as a *witness* of the judgement; Γ and Δ carry the types of the free connectors in P, as unordered sets. There is no notion of type for P itself, instead the derivable statement shows how P is connectable. In fact, this notion of type assignment on $\mathcal{X}$-circuits has been compared to that on *processes* of the π-calculus[2]. The following result was shown in [9]:

Theorem 5.1.4 (Witness reduction [9]) *If $P :\cdot \Gamma \vdash \Delta$, and $P \to Q$, then $Q :\cdot \Gamma \vdash \Delta$.*

In [85] a notion of *principal contexts* is defined by providing a sound and complete algorithm $\mathcal{W}^{\mathcal{X}}$ that, given a circuit P, returns a pair of contexts $\langle \Gamma, \Delta \rangle$ typing the free connectors of P. It was also shown that P is then a witness for $\Gamma \vdash \Delta$, and the pair of contexts is indeed the most general.

5.2 Building Curry-Howard Correspondences

In reviewing existing works of those seeking Curry-Howard correspondences between formal logics and computational calculi, a general 'recipe' for building such a correspondence can be seen. We outline below only what we consider to be the *key steps* involved.

1. Build a propositional language from a set of propositional variables and propositional formulas built from *primitive* logical connectives.
2. Decide on a framework to *prove* the validity of statements in the language, and formally (i.e., analytically) define each connective in the framework using inference rules.

[2]The relation between $\mathcal{X}$ and π, and the implication of that relation on the connection between Classical Logic and π, is the subject of ongoing research; the first results have been reported on in [8].

3. Determine a syntax of terms, and a method of inhabiting proofs with terms.
4. Decide on the set of 'normal' proofs, together with a normalisation procedure and extract the reduction rules of the term calculus from the transformations prescribed by the normalisation procedure.
5. If desired, erase any type information from the term syntax and study the computational behaviour of the corresponding term calculus.

The important point to note about the above recipe is that, using this approach, any reduction behaviour is determined by the logic. The types may be erased, but the shape of the reduction rules remain the same.

In the following subsections we will motivate and propose the specific design choices we have taken in our research to obtain computational term calculi which hold Curry-Howard correspondences with a variant of Gentzen's formulation of Classical Logic, while following the above recipe.

We will address the choices of primitive connectives that the propositional language is based on (step one of the recipe) in a separate section (Section 5.3) since it is an open-ended topic. We will address the remaining steps in the following three subsections, beginning with setting up a logical framework.

5.2.1 A Sequent Calculus Framework

As motivated in the background section (Section 2.2), we will work with sequent calculus formulations of classical logic that adhere to Gentzen's notion of logical consequence (Definition 2.2.3); this framework, which we called G3A-BASIC, was given in Definition 2.2.6.

When extending the framework with a logical connective, suitable proof rules need to be defined. We will work with invertible formulations of inference rules for logical connectives (Definition 2.2.7), i.e., for each connective a single pair of sequent calculus inference rules will be defined that introduce a formula with that principal connective on the left and right-hand side of the turnstile of each rule's conclusion.

It may not always be the case that a set of suitable sequent proof rules for a particular connective are obvious. Fortunately, there exist sets of arity-two connectives that are *functionally complete*, that is, *every* possible logical connective can be defined in terms of the members of that set. For example, elements of the functionally complete set of connectives $\{\vee, \neg\}$ can be composed in various ways to

define any logical connective, as can the elements of $\{\uparrow\}$ (corresponding to the set containing the 'nand' connective). Furthermore, some functionally complete sets of arity two contain connectives whose inference rules are already known (e.g., $\{\wedge, \neg\}$). Using this result, one can derive suitable sequent rules (which we will generically call $(\complement_i^n L)$ and $(\complement_i^n R)$) for an arbitrary connective $\complement_i^n$ by choosing a formula F logically equivalent to $\complement_i^n$ built up from elements of the functionally complete set (which will have known inference rules). Now, by constructing (partial) derivation schemes which introduce F on the left and right-hand side of the turnstile, we can obtain the inference rules for $\complement_i^n$ by translating all sub-derivations to be completed to sub-proofs, and replacing the formula F in the endsequent with the formula $\complement_i^n(A_1, \ldots, A_n)$. We illustrate this process in the following example.

Example 5.2.1 (Deriving Sequent rules for Difference) *The set $\{\rightarrow, \neg\}$ is functionally complete. Using truth tables, we can express the difference connective $\complement^2_{0010_2}(A, B)$ (for which we adopt the shorthand $A-B$), in terms of negation and implication. We find that,*

$$A-B \equiv \neg A \rightarrow B$$

Using the proof rules for $\rightarrow$ and $\neg$, we can build two (incomplete) derivations that introduce the formula $\neg A \rightarrow B$ to the left and right hand sides of a sequent. From this derivation, we can extract the proof rules $(-L)$ and $(-R)$ as follows:

$$\dfrac{\dfrac{\begin{array}{cc}\square & \square \\ \Gamma \vdash A, \Delta & \Gamma, B \vdash \Delta\end{array}}{\Gamma \vdash A \rightarrow B, \Delta}\,(\rightarrow L)}{\Gamma \vdash \neg(A \rightarrow B), \Delta}\,(\neg R) \qquad \textit{gives} \qquad \dfrac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma \vdash A - B, \Delta}\,(-R)$$

$$\dfrac{\dfrac{\dfrac{\square}{\Gamma, A \vdash B, \Delta}\,(\rightarrow R)}{\Gamma \vdash A \rightarrow B, \Delta}}{\Gamma, \neg(A \rightarrow B) \vdash \Delta}\,(\neg L) \qquad \textit{gives} \qquad \dfrac{\Gamma, A \vdash B, \Delta}{\Gamma, A - B \vdash \Delta}\,(-L)$$

5.2.2 Generating Term Syntax

Step three of the recipe requires the inhabitation of proof rules with term syntax. We choose to follow the style of inhabitation used in $\mathcal{X}$, since it best expresses the structure of the proof, while preserving symmetries of the logic at the level of the syntax.

When deriving the syntax to represent a particular proof rule, we will reformulate the contexts of G3A-BASIC (and extensions of G3A-BASIC) to contexts of sockets and contexts of plugs, and adhere to Urban's context convention. Formulas which occur on the left of a sequent will be labelled with sockets $x, y, z, \ldots$ while formulas on the right will be labelled with plugs $\alpha, \beta, \gamma, \ldots$. Any sub-proofs present in the rule will be represented as sub-terms of the syntax. Formulas which disappear from such sub-proofs by application of the proof rule (in a sense they are 'bound' by the rule) will correspond to bound connectors on the sub-terms, while a new formula which is introduced by the rule corresponds to a free connector of the appropriate kind.

We will adopt a generic naming convention for inference rules for arbitrary connectives, and the circuits that correspond to those rules. For an arbitrary connective of arity n, C_i^n, the left and right inference rules are called $(\mathsf{C}_i^n L)$ and $(\mathsf{C}_i^n R)$, respectively. The circuit constructors that inhabit these rules called respectively *input circuits* and *output circuits*; these will be written as $\mathsf{C}_i^n I$ and $\mathsf{C}_i^n O$.

The term constructors generated by the above process will be constrained to adhere to the following definition.

Definition 5.2.2 (Generic Circuit Grammar) *The circuits of the $\mathcal{X}$-style calculi are defined by the following grammar, where x ranges over the infinite set of* sockets*, and α over* plugs*.*

P, Q	$::=$	$\langle x \cdot \alpha \rangle$	capsule
	$\mid$	$P\widehat{\alpha} \dagger \widehat{x}Q$	cut
	$\mid$	$x \cdot [\overline{B}]$	input circuit
	$\mid$	$[\overline{B}] \cdot \alpha$	output circuit
$\overline{B}$	$::=$		empty
	$\mid$	*blocks*	one or more blocks
blocks	$::=$	*block*	one block
	$\mid$	*block* , *blocks*	more than one block
block	$::=$	$\overline{\mathtt{S}}\ P$	block with only bound socket(s)
	$\mid$	$P\ \overline{\mathtt{P}}$	block with only bound plug(s)
	$\mid$	$\overline{\mathtt{S}}\ P\ \overline{\mathtt{P}}$	block with both bound socket(s) and bound plug(s)

$\overline{\mathrm{s}} ::= \hat{x}$ a socket
$\quad\mid\ \hat{x}\overline{\mathrm{s}}$ more than one socket

$\overline{\mathrm{p}} ::= \hat{\alpha}$ a plug
$\quad\mid\ \overline{\mathrm{p}}\hat{\alpha}$ more than one plug

Where many blocks are used to build a circuit, we will apply an ordering over the blocks according to the number of sockets and plugs that are bound. The convention applied is as follows:

1. *blocks with no bound sockets are at the front of the list, in decreasing order of the number of plugs binding over the sub-term, e.g., the sequence: …, $P\widehat{\alpha}\widehat{\mu}\widehat{\pi}$, $P\widehat{\alpha}\widehat{\mu}$, $P\widehat{\alpha}$.*
2. *blocks with no bound plugs are at the end of the list, in increasing order of the number of sockets binding over the sub-term, e.g., the sequence: $\widehat{x}P$, $\widehat{x}\widehat{y}P$, $\widehat{x}\widehat{y}\widehat{z}P$, …*
3. *blocks with both bound sockets and bound plugs are placed in between the sublists defined by the first two parts, and are ordered increasingly by the number bound sockets, and within each group with the same number of sockets but different number of plugs, ordered decreasingly by the number of bound plugs, e.g., the sequence: $\widehat{x}P\widehat{\alpha}$, $\widehat{y}\widehat{x}P\widehat{\alpha}\widehat{\mu}$, $\widehat{y}\widehat{x}P\widehat{\alpha}$, $\widehat{z}\widehat{y}\widehat{x}P\widehat{\alpha}\widehat{\mu}$, $\widehat{z}\widehat{y}\widehat{x}P\widehat{\alpha}$, …*

Under the above construction, the export $\widehat{x}P\widehat{\beta}\cdot\alpha$ and the import $P\widehat{\alpha}\,[y]\,\widehat{x}Q$ are syntactic sugar for the output and input circuits $[\widehat{x}P\widehat{\beta}]\cdot\alpha$ and $y\cdot[P\widehat{\alpha}, \widehat{x}Q]$ (their generic names are $\mathsf{C}^2_{1101_2}O$ and $\mathsf{C}^2_{1101_2}I$).

Below we give an example of how an $\mathcal{X}$-style circuit can be derived from an inference rule.

Example 5.2.3 (Annotating the Rule $(\rightarrow L)$) *In this example, we will extract the input circuit $\rightarrow I$ from the $(\rightarrow L)$ proof rule so that the syntax adheres to Definition 5.2.2.*

We begin with the logical rule for introducing a formula whose principal connective is implication to the antecedent part of a sequent.

$$\frac{\Gamma \vdash \Delta, A \qquad B, \Gamma \vdash \Delta}{A \rightarrow B, \Gamma \vdash \Delta}\,(\rightarrow L)$$

Each formula in each rule premise is annotated with a connector. We annotate the formula A with a plug, say α, since it appears on the right of the turnstile, and the formula B with

a socket x since it appears on the left. We also annotate the formula $A \to B$ with a socket y since it appears on the left of the turnstile.

In an application of the proof rule, the premise sequents will map to sub-proofs that derive (an instantiation of) each premise sequent. We introduce circuit variables (P and Q respectively) to act as witnesses for each of these sub-proofs.

With regards to binders, each premise of the rule $(\to L)$ discharges a formula, which becomes bound to the respective sub-proofs. This is mirrored in the term syntax by binding the appropriate connector to its sub-circuits, i.e., α is bound in P and x is bound in Q. This gives the rule:

$$\frac{P :\cdot \Gamma \vdash \Delta, \alpha{:}A \qquad Q :\cdot x{:}B, \Gamma \vdash \Delta}{y \cdot [P\widehat{\alpha}, \widehat{x}Q] :\cdot y{:}A \to B, \Gamma \vdash \Delta}\ (\to L)$$

We point out a small subtlety related to the implicit formulation of contraction rules when moving from a (usual) sequent calculus which whose contexts are sets of *formulas* (such as G3A-BASIC) to a sequent calculus whose contexts are sets of $\langle$*label, formula*$\rangle$ pairs (such as *G3a′*). Consider the following derivation in G3A-BASIC.

$$\frac{\boxed{}}{\vdash A \to B}\ (\to R)$$

In the above, notice the implicit contraction of the formula $A \to B$ when the rule $(\to R)$ is applied. We would represent the above in *G3a′* using:

$$\frac{\boxed{\phantom{x{:}A \vdash \alpha{:}A \to B, \beta{:}B}}}{\vdash \alpha{:}A \to B}\ (\to R)$$

Here, although the application of $(\to R)$ introduces a new formula with principal connective $\to$ to the conclusion, we make sure to *choose* the same label as the existing $A \to B$ formula (i.e., α) in order to represent the contraction.

A slightly more complicated example is the representation the following G3A-BASIC proof in *G3a′*:

$$\frac{\boxed{} \quad \boxed{}}{\vdash B}\ (Cut)$$

In the above, the cut formula A is eliminated from the derivation, but additionally the two formulas B in the left and right derivations are contracted. In the

corresponding proof $G3a'$ proof, if B shares the label across the sub-proofs, there is no problem and we can simply merge the two contexts. For example,

$$\dfrac{\vdash \alpha{:}B, \gamma{:}A \qquad x{:}A \vdash \alpha{:}B}{\vdash \alpha{:}B}\,(Cut)$$

However, if the subproofs are:

$$\vdash \alpha{:}B, \gamma{:}A \qquad \textit{and} \qquad x{:}A \vdash \delta{:}B$$

then applying (Cut), will build the conclusion $\vdash \alpha{:}B, \delta{:}B$, and so the contraction step is not reproduced. In this case, one should introduce an *extra* cut with an appropriate instance of the axiom rule to perform the renaming as needed. There is in fact a choice of two $G3a'$ derivations which can be mapped onto; these are shown below.

$$\dfrac{\dfrac{\vdash \alpha{:}B, \gamma{:}A \qquad \dfrac{}{y{:}B \vdash \delta{:}B}\,(Ax)}{\vdash \delta{:}B, \gamma{:}A}\,(Cut) \qquad x{:}A \vdash \delta{:}B}{\vdash \delta{:}B}\,(Cut)$$

$$\dfrac{\vdash \alpha{:}B, \gamma{:}A \qquad \dfrac{x{:}A \vdash \delta{:}B \qquad \dfrac{}{y{:}B \vdash \alpha{:}B}\,(Ax)}{x{:}A \vdash \alpha{:}B}\,(Cut)}{\vdash \alpha{:}B}\,(Cut)$$

Essentially we have used an application of the cut and axiom to *rename* a labelled formula in one of the sub-proofs.

5.2.3 Normalisation and Reduction Rules

Recall that the normal proofs in sequent calculi are cut-free proofs. In general, a local cut-elimination procedure for sequent calculus proofs consists of applying permutations to the proof to shift the cut towards the leaves of the derivation. At various points, the complexity of the cut-formula may be reduced by applying a principal logical rule. This process was described in some detail in Section 2.2.3. The exact choice of permutations to apply is a topic which has been studied in detail by a great many number of authors (e.g., [86, 34, 61, 41]).

Due to our choice of syntax, and familiarity with the $\mathcal{X}$-calculus, we will choose

to work with a generalisation of the $\mathcal{X}$'s reduction system (though we could have equally chosen Urban's procedure [86] or the colouring annotations of Danos *et al.* [34], etc.).

In $\mathcal{X}$-calculi, normal circuits are those built without using the cut circuit (i.e., $M\widehat{\alpha} \dagger \widehat{x}N$). In this section we will outline the key ingredients for a general normalisation of $\mathcal{X}$-style circuits, corresponding to a local cut-elimination procedure on proofs.

Whichever logical connectives are employed, we will always keep a basic set of reduction rules which deal with cuts and capsules.

Definition 5.2.4 (Basic Reduction Rules, $\mathcal{R}$) *The set of* basic reduction rules, $\mathcal{R}$, *consists of the following rules from Definitions 3.1.4 and 3.1.7,*

$(cap\text{-}rn)$	renaming of a capsule
$(act\text{-}\mathrm{L}), (act\text{-}\mathrm{R})$	activation of a cut
$(\not\sim d), (d \not\sim)$	deactivation of a cut
$(cap \not\sim), (\not\sim cap)$	garbage-collection of a cut
$(cut \not\sim), (\not\sim cut)$	propagation of an active cut through an inactive cut

The notion of a plug or socket being *introduced* can be extended to the case of a generic circuit: the circuit P introduces x (respectively, α) iff x is free in P but not in any of its proper sub-circuits.

Reduction rules for calculi built in the style of $\mathcal{X}$ are either propagation rules or logical rules. The propagation rules formalise the substitution-like operations $P\{\alpha \leftrightarrow \widehat{x} Q\}$ and $Q\{P\widehat{\alpha} \leftrightarrow x\}$, which, as discussed in Section 3.1, place copies of cuts next to sub-circuits *introducing* the connector involved in the 'substitution'.

The general approach is to push copies of the cut *ins*ide the sub-terms, leaving a copy on the *outs*ide if an occurrence of the desired connector was present at this level (c.f., $\not\sim imp\text{-}outs$). The appropriate rules for propagation over a construct which introduces a plug may be derived symmetrically. The rules that describe this process for an arbitrary connective are given in the following definition.

Definition 5.2.5 (Propagation Rules for $\binom{n}{i} I$ and $\binom{n}{i} O$) *The input and output circuits that correspond to an inference rule can, in general, be written as:*

$$x \cdot [\overrightarrow{u_1} P_1 \overrightarrow{\pi_1}, \ldots, \overrightarrow{u_s} P_s \overrightarrow{\pi_s}] \quad \textit{and} \quad [\overrightarrow{v_1} Q_1 \overrightarrow{\sigma_1}, \ldots, \overrightarrow{w_1} Q_t \overrightarrow{\sigma_t}] \cdot \alpha$$

where, $s+t \leq 2^n$, P_i, Q_i are circuits (with $i \leq n$) and the notation $\overrightarrow{\cdot}$ stands for a sequence of zero or more bound connectors of a particular kind, such that $\overrightarrow{v_i}\, Q_i\, \overrightarrow{\sigma_i}$ is a block (see Definition 5.2.2).

For a pair of generic input and output circuits (shown above), the following six *propagation rules need to be added to the calculus, (where k, δ are* fresh*),*

1. *In case the connector of the cut matches the single free connector of the generic circuit, deposit a fresh inactive cut introducing that connector on the* outside*, and build an active cut with each proper sub-circuit. The corresponding rules $(\nwarrow \complement_i^n I\text{-}outs)$ and $(\complement_i^n O\text{-}outs \nearrow)$ are:*

$$\begin{array}{ll} R\widehat{\gamma} \nwarrow \widehat{y}(y\cdot[\overrightarrow{u_1}P_1\overrightarrow{\pi_1}, \ldots, \overrightarrow{u_s}P_s\overrightarrow{\pi_s}]) & \rightarrow R\widehat{\gamma} \dagger \widehat{k}(k\cdot[\overrightarrow{u_1}(R\widehat{\gamma} \nwarrow \widehat{y}P_1)\overrightarrow{\pi_1}, \ldots, \overrightarrow{u_s}(R\widehat{\gamma} \nwarrow \widehat{y}P_s)\overrightarrow{\pi_s}]) \\ ([\overrightarrow{v_1}Q_1\overrightarrow{\sigma_1}, \ldots, \overrightarrow{w_1}Q_t\overrightarrow{\sigma_t}]\cdot\gamma)\widehat{\gamma} \nearrow \widehat{y}R & \rightarrow ([\overrightarrow{v_1}(Q_1\widehat{\gamma} \nearrow \widehat{y}R)\overrightarrow{\sigma_1}, \ldots, \overrightarrow{v_t}(Q_t\widehat{\gamma} \nearrow \widehat{y}R)\overrightarrow{\sigma_t}]\cdot\delta)\widehat{\delta} \dagger \widehat{y}R \end{array}$$

2. *If there is no match between the connector of the cut and the outer connector of the generic circuit, place active cuts '*in*side' the generic circuit (i.e., build a cut with each proper sub-circuit). The corresponding rules $(\nwarrow \complement_i^n I\text{-}ins)$ and $(\complement_i^n O\text{-}ins \nearrow)$ are:*

$$\begin{array}{ll} R\widehat{\gamma} \nwarrow \widehat{y}(x\cdot[\overrightarrow{u_1}P_1\overrightarrow{\pi_1}, \ldots, \overrightarrow{u_s}P_s\overrightarrow{\pi_s}]) & \rightarrow x\cdot[\overrightarrow{u_1}(R\widehat{\gamma} \nwarrow \widehat{y}P_1)\overrightarrow{\pi_1}, \ldots, \overrightarrow{u_s}(R\widehat{\gamma} \nwarrow \widehat{y}P_s)\overrightarrow{\pi_s}] \quad \leftarrow x \neq y \\ ([\overrightarrow{v_1}Q_1\overrightarrow{\sigma_1}, \ldots, \overrightarrow{w_1}Q_t\overrightarrow{\sigma_t}]\cdot\alpha)\widehat{\gamma} \nearrow \widehat{y}R & \rightarrow [\overrightarrow{v_1}(Q_1\widehat{\gamma} \nearrow \widehat{y}R)\overrightarrow{\sigma_1}, \ldots, \overrightarrow{v_t}(Q_t\widehat{\gamma} \nearrow \widehat{y}R)\overrightarrow{\sigma_t}]\cdot\alpha \quad \leftarrow \gamma \neq \alpha \end{array}$$

3. *If the free connector of the generic circuit is of the opposite 'kind' to the active cut, simply build active cuts with the proper sub-circuits of the generic circuit. The corresponding rules $(\complement_i^n I \nearrow)$ and $(\nwarrow \complement_i^n O)$ are:*

$$\begin{array}{ll} (x\cdot[\overrightarrow{u_1}P_1\overrightarrow{\pi_1}, \ldots, \overrightarrow{u_s}P_s\overrightarrow{\pi_s}])\widehat{\gamma} \nearrow \widehat{y}R & \rightarrow x\cdot[\overrightarrow{u_1}(P_1\widehat{\gamma} \nearrow \widehat{y}R)\overrightarrow{\pi_1}, \ldots, \overrightarrow{u_s}(P_s\widehat{\gamma} \nearrow \widehat{y}R)\overrightarrow{\pi_s}] \\ R\widehat{\gamma} \nwarrow \widehat{y}([\overrightarrow{v_1}Q_1\overrightarrow{\sigma_1}, \ldots, \overrightarrow{w_1}Q_t\overrightarrow{\sigma_t}]\cdot\alpha) & \rightarrow [\overrightarrow{v_1}(R\widehat{\gamma} \nwarrow \widehat{y}Q_1)\overrightarrow{\sigma_1}, \ldots, \overrightarrow{v_t}(R\widehat{\gamma} \nwarrow \widehat{y}Q_t)\overrightarrow{\sigma_t}]\cdot\alpha \end{array}$$

Of course rule names can be changed; the rule names we presented above ensures no conflict with rule names of other logical connectives.

This leaves us with having to define the appropriate logical reduction rules for the connective. These logical rules can be grouped into two sets: (i) those which perform a renaming on an introduced connector, and (ii) the 'principal' reduction rule(s) which define(s) the main computational behaviour of the connective. The first can be defined as follows.

Definition 5.2.6 (Renaming Rules for $\complement_i^n I$ and $\complement_i^n O$) *The circuits $\complement_i^n I$ and $\complement_i^n O$ can, in general, be written as $x\cdot[I_1, \ldots, I_s]$ and $[O_1, \ldots, O_t]\cdot\alpha$, where I, O ranges over blocks, and $s+t \leq 2^n$. The extension of the syntax with these circuits requires that the following renaming logical rules need to be added to the term calculus.*

$$\begin{array}{lll} (\complement_i^n I\text{-}rn)\text{:} & \langle x\cdot\gamma\rangle\widehat{\gamma} \dagger \widehat{y}(y\cdot[I_1, \ldots, I_s]) & \rightarrow x\cdot[I_1, \ldots, I_s] \qquad \leftarrow y \text{ introduced} \\ (\complement_i^n O\text{-}rn)\text{:} & ([O_1, \ldots, O_t]\cdot\gamma)\widehat{\gamma} \dagger \widehat{y}\langle y\cdot\alpha\rangle & \rightarrow [O_1, \ldots, O_t]\cdot\alpha \qquad \leftarrow \gamma \text{ introduced} \end{array}$$

A logical rule must also be defined to show how a cut whose cut formula is introduced by a connective's left and right logical rules can be eliminated from a proof. We call the corresponding proof transformation the *principal reduction rule* for the connective (with generic rule name (C_i^n)).

Perhaps the most straightforward way of building a principal reduction rule for a arbitrary logical connective C_i^n would be to first express the connective as an equivalent formula built using a combination of some well studied connectives that form a functionally complete set (e.g., the set containing the connectives negation and disjunction). By building derivation schemes of the equivalent formulation on the left and right of a sequent scheme, one can obtain the inference rules for that connective. Now by considering the cut-elimination of the derivation scheme built by applying (Cut) to the right and left premises respectively, one can obtain a derivation scheme whose structure corresponds to the right-hand side of the principal reduction rule for C_i^n. We use this approach in Section 5.5 where we construct a calculus $\mathcal{X}^{\leftrightarrow}$ with a Curry-Howard correspondence with a sequent calculus employing the if-and-only-if connective as a primitive (since the principal reduction rule for the connective turns out to be non-trivial to derive directly).

Finally, we remark that if one employs more than one logical connective in the logic, the cut rule is not applicable between different connectives. In the corresponding untyped term calculus, it is permitted for cuts to be formed between circuits corresponding to different logical rules. In this work, we will consider such cuts to be irreducible. Therefore, when more than one logical connective is employed, the notion of normal form is extended; in particular it will be possible to have untypeable normal forms which contain cuts.

We will say that a term calculus extracted from a sequent calculus following the methods described in this section are built *in the style of* $\mathcal{X}$.

5.3 Relating Binary Logical Connectives

In the previous section we outlined a general 'recipe' which can be followed to construct a Curry-Howard pair of calculi. In this section we will address the first step of the 'recipe': deciding on which connectives to consider as primitives in a logic.

It is interesting to ask why different authors have based their logics on partic-

ular choices of primitive connectives. Implication is the most popular choice of connective, presumably because it is well understood that its computational behaviour is related to function abstraction and application. More adventurous authors have sought to define Curry-Howard pairs of calculi starting from logics which feature other logical connectives as primitives. For example, Wadler's Dual Calculus [91] features primitive conjunction, disjunction and negation, while Crolard uses the 'difference' connective in his formulae-as-types notion of subtractive logic [31].

It is rare to find investigations into Curry-Howard correspondences that employ logics built from connectives of arity greater than two (although for an example, see [64]). The likely reason for this is the existence of functionally complete sets of connectives. However, we note that it is also common practice to augment functionally complete sets of connectives with additional primitive connectives. The reason for this is that a certain amount of clarity is gained from using a larger set of primitives; consider the extreme case of basing a computational calculus solely on the 'nand' connective.

We notice curious biases in the field towards particular logical connectives (i.e., and, or, implication, negation), which naturally prompts us to ask why the remaining twelve connectives of arity two are rarely studied. In joint work with Summers [74], we investigated *all* of the arity-two classical logical connectives and focused on relationships that existed between. This section is a summary of our work.

We began our investigations with the following questions:

(a) How many logical connectives are there of arity n ($n \geq 0$)?
(b) How many of these depend on all n inputs (we say these have *true arity* n, Definition 2.2.13)?
(c) How many of these *always* depend on all n inputs?

Following some routine counting exercises, we determined the following result.

Theorem 5.3.1 (Enumerating Logical Connectives [74]) *For any integer $n \geq 0$:*

1. *There are 2^{2^n} logical connectives of arity n.*
2. *The number of these which depend on all n inputs (those of true arity n), $t(n)$ is given by the following formula: $t(n) = 2^{2^n} - \sum_{i=0}^{n-1} \binom{n}{i} t(i)$.*

3. *There are exactly two connectives of arity n which always depend on all n inputs; these are the parity function (which returns true exactly when an even number of its arguments are), and its negation.*

Regarding the connectives whose arity corresponds to their true arity (part *(b)* above), we can observe that for the case:

$t(0) = 2$: The two connectives are the logical constants $\top$ and $\bot$, which can be seen as connectives of arity 0.

$t(1) = 2$: The connectives are the *identity connective* (which returns its input argument unchanged) and the *negation connective* (which returns the negation of its input argument).

$t(2) = 10$: Considering there are 16 arity-two connectives, we subtract from this set the following six connectives (whose true arity is not 2):

- the connective that ignores both inputs, always returning true.
- the connective that ignores both inputs, always returning false.
- the two connectives which always return the value of one of the two inputs (ignoring the other input).
- the two connectives which always returns the negation of value of one of the two inputs (ignoring the other input).

Using the generic notation for describing arbitrary classical logical connectives introduced in Definition 2.2.11, we list the 2^{2^2} logical connectives of arity two in Figure 5.1(a). We also mention the standard name and symbol associated with each connective where it exists. Taking this set of arity two connectives, we attempted to determine which of them could be 'obtained' from others by applying some simple translations involving negation. We defined a notion of 'obtainability' as follows.

Definition 5.3.2 (Obtainability) *For any two binary connectives C_i^2,C_j^2, and for all assignments of truth values to the propositional variables A, B:*

Reversal *: We say C_i^2 is the* reverse *of C_j^2 iff $\mathsf{C}_i^2(A,B) \equiv \mathsf{C}_j^2(B,A)$.*

Duality *: We say C_i^2 is the* dual *of C_j^2 iff $\mathsf{C}_i^2(A,B) \equiv \neg\mathsf{C}_j^2(\neg A,\neg B)$.*

Negation *: We say C_i^2 is the* negation *of C_j^2 iff $\mathsf{C}_i^2(A,B) \equiv \neg\mathsf{C}_j^2(A,B)$.*

Flipping inputs *: We say C_i^2 is obtained from C_j^2 by* flipping an input *if either*

$$\mathsf{C}_i^2(A,B) \equiv \mathsf{C}_i^2(\neg A,B) \quad \textit{or} \quad \mathsf{C}_i^2(A,B) \equiv \mathsf{C}_i^2(A,\neg B)$$

We say we can obtain C_i^2 *from* C_j^2 *iff* C_j^2 *can be defined in terms of* C_i^2 *by applying one or more of the above relations.*

The above relations are illustrated in Figure 5.1(b) as intuitively labelled arrows. Notice that the relations partition the connectives into five groups, where each connective in a particular group is logically expressible using another connective in that group.

This observation led us to believe that it should be possible to determine the pair of inference rules for each connective in a group by applying some basic transformations to the inference rules of another connective. We give a summary of our findings below.

Reversal : The reversal operation, which simply swaps the order of the supplied arguments, has no effect on the shape of the inference rules for the connective. Consider the reverse of the implication connective, which we shall write $\leftarrow$, whose pair of sequent rules are shown below.

$$\frac{\Gamma \vdash \Delta, B \quad A, \Gamma \vdash \Delta}{A \leftarrow B, \Gamma \vdash \Delta}\,(\leftarrow L) \qquad \frac{B, \Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A \leftarrow B}\,(\leftarrow R)$$

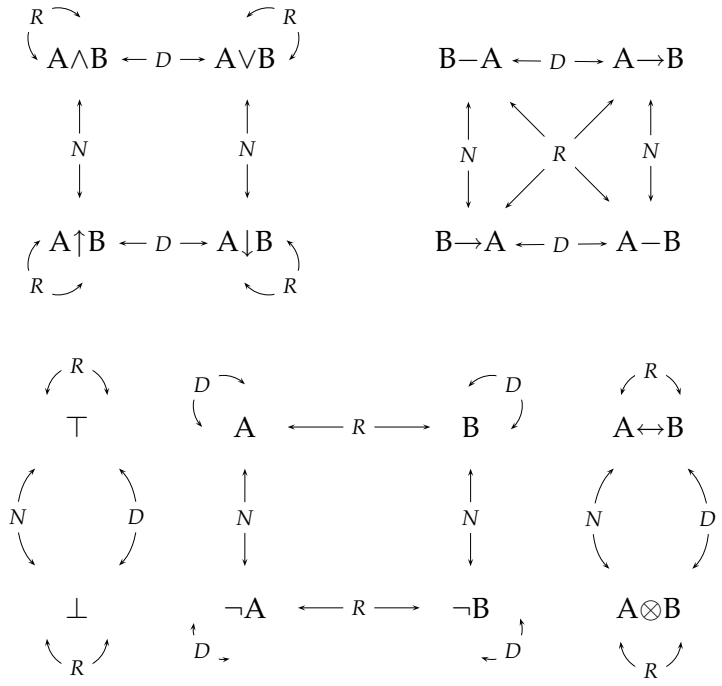
Using the method of generating term annotations detailed in Section 5.2.2, we find that the inhabitants of these rules are exactly the same as those for implication. This is because the same inputs and outputs are bound and introduced by the inference rules, the only difference being the positioning of A and B, which is irrelevant since types are not part of our term syntax. The associated cut-elimination rules will also be exactly the same as those for implication, and therefore so will the computational content obtained.

As a result of this observation, we choose to examine the connectives in question modulo reversals. Since most of the connectives in Figure 5.1(b) are symmetrical (remain the same when reversed), this actually only reduces the number of connectives in question by four. Our notation becomes rather less cumbersome, in that we need not write formulas to define any of the connectives (e.g., $B-A$ was used to write the reverse of $A-B$); we can now write an unambiguous symbol for each. This is shown in Figure 5.1(c).

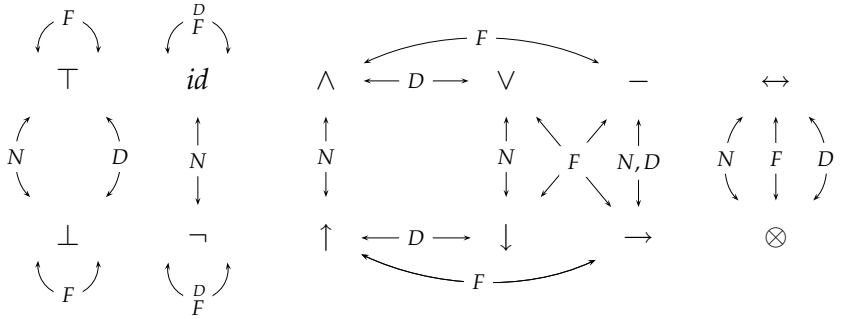
Negation : Given the inference rules for an arbitrary connective, it is straightforward to derive suitable rules for the negation of that connective. For example, the negation of implication ($\rightarrow$) is the 'difference' connective ($-$), and by seeking suitable derivations for the formula $\neg(A \rightarrow B)$ on both the

$\mathbf{C}^2_{0000_2}$	$\top$	truth	$\mathbf{C}^2_{1000_2}$	$\downarrow$	nor
$\mathbf{C}^2_{0001_2}$	$\wedge$	conjunction	$\mathbf{C}^2_{1001_2}$	$\leftrightarrow$	if-and-only-if
$\mathbf{C}^2_{0010_2}$	$-$	difference	$\mathbf{C}^2_{1010_2}$	$\neg_B$	negation (of second argument)
$\mathbf{C}^2_{0011_2}$	id_A	first argument	$\mathbf{C}^2_{1011_2}$		*reverse of implication*
$\mathbf{C}^2_{0100_2}$		*reverse of difference*	$\mathbf{C}^2_{1100_2}$	$\neg_A$	negation (of first argument)
$\mathbf{C}^2_{0101_2}$	id_B	second argument	$\mathbf{C}^2_{1101_2}$	$\rightarrow$	implication
$\mathbf{C}^2_{0110_2}$	$\otimes$	exclusive-or	$\mathbf{C}^2_{1110_2}$	$\uparrow$	nand
$\mathbf{C}^2_{0111_2}$	$\vee$	disjunction	$\mathbf{C}^2_{1111_2}$	$\bot$	falsehood

(a) The Sixteen Arity-Two Connectives



(b) Relations between the connectives



(c) Arity-Two Connectives "Modulo Reversals"

Figure 5.1: Boolean Connectives of Arity Two

left and the right of a sequent, one can derive the appropriate rules for 'difference' as shown below.

$$\frac{A, \Gamma \vdash \Delta, B}{A - B, \Gamma \vdash \Delta}\,(-L) \qquad \frac{\Gamma \vdash \Delta, A \qquad B, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, A - B}\,(-R)$$

The input and output circuits that inhabit the above rules are $x \cdot [\widehat{y} R \widehat{\beta}]$ and $[P\widehat{\beta}, \widehat{x} Q] \cdot \alpha$ respectively[3]. Notice that these terms have the same lists of blocks in common with the terms for implication; the only difference is that the free connector introduced appears on the opposite side of the sequent (due to the negation). This generalises to any connective and its negation; the term representations will be identical for each, but with the left and right free connectors exchanged. Furthermore, in defining a cut-elimination rule, one can see that the reduct of the key logical rule will be the same in the cases of $\rightarrow$ and $-$, and in general for a connective and its negation. These ideas also generalise to any connective and its negation.

Duality : The duality operation has the effect of 'swapping the side' of *every* formula in the proof, since both the arguments and the connective itself are negated. For example, compare the rules for the pair of 'dual' connectives conjunction $\wedge$ and disjunction $\vee$:

$$\frac{A, B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta}\,(\wedge L) \qquad \frac{\Gamma \vdash \Delta, A \qquad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B}\,(\wedge R)$$

$$\frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \vee B}\,(\vee R) \qquad \frac{A, \Gamma \vdash \Delta \qquad B, \Gamma \vdash \Delta}{A \vee B, \Gamma \vdash \Delta}\,(\vee L)$$

Flipping of an Input : The effect of flipping an input is to negate only *one* of the inputs to a connective, which in turn corresponds to the bound occurrences of one of the formulas swapping sides in the rules. For example, implication can be obtained from disjunction by flipping the first input ($A \rightarrow B \equiv \neg A \vee B$). One can see this also by comparing the sequent rules of implication (below) to those for disjunction (above).

$$\frac{A, \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \rightarrow B}\,(\rightarrow R) \qquad \frac{\Gamma \vdash \Delta, A \qquad B, \Gamma \vdash \Delta}{A \rightarrow B, \Gamma \vdash \Delta}\,(\rightarrow L)$$

As can be seen from Figure 5.1(c), the above transformations can be used to relate six of the arity two connectives. In Section 5.4, we will see that the computa-

[3]The sugared syntax would be $x \cdot \widehat{y} R \widehat{\beta}$ and $P\widehat{\beta}\,[\alpha]\,\widehat{x} Q$.

tional behaviour of each of these connectives can be associated with some kind of 'pairing' functionality.

The remaining connectives come in related groups of two. The negation and identity connectives (which are really of true arity one) have a computational behaviour very different from the group of six connectives. The negation connective is traditionally associated with behaviour relating to the manipulation of continuations. The identity connective can be seen to have a very trivial computational content (at best it provides a kind of *aliasing*, where a connector is bound within a subterm and then immediately exported again with a new name).

The $\top$ and $\bot$ connectives are rather unusual, since it turns out they each have no sensible proof rule for introducing the connective on one side of the sequent (in fact a rule *can* be added but it amounts to a special case of weakening). In the case of $\top$, there is only a sensible rule for introduction on the right, and symmetrically $\bot$ only has an introduction rule on the left. These rules are given below:

$$\frac{}{\Gamma, \vdash \top, \Delta}\,(\top R) \qquad\qquad \frac{}{\Gamma, \bot \vdash \Delta}\,(\bot L)$$

Since these rules introduce a new formula without binding any existing ones, they can be seen to be inhabited by terms which make available an output (respectively input) which isn't connected to anything. As far as reduction rules are concerned, it is impossible to add the usual principal logical rule, since there is no pair of left and right terms to connect. When one considers a cut between (for example) a $(\top R)$ rule on the left and some other term in the right, it is clear that the connector bound on the other side of the cut must be introduced by weakening (if the cut is typeable). In this way the terms to represent $\top$ and $\bot$ can be used to provide 'dead-end' cuts, which when evaluated simply disappear (c.f., garbage collection, Lemma 3.1.21).

There remain only two binary connectives to discuss, being $\leftrightarrow$ ('if-and-only-if') and $\otimes$ ('exclusive or'). As can be seen from Figure 5.1(c), these two are *obtainable* only from each other. The (similar) operations they describe are difficult to relate directly to any of the other connectives since there are no 'simple' equivalent formulas which express these connectives in terms of the others. We showed in [74, Thm. 5.3] that any formula equivalent to $\leftrightarrow$ or $\otimes$ (not constructed using $\leftrightarrow$ or $\otimes$) must duplicate at least one of the supplied arguments. The result suggested that the two connectives $\leftrightarrow$ and $\otimes$ may have some interesting complexity which the other binary connectives do not. Subsequently, we decided to investigate the computational content of these two connectives, which appears not to have

been attempted so far in the literature. In particular, no cut-elimination rule (or analogously, proof reduction rule in a Natural Deduction setting) seems to have been defined for these connectives. Our investigations into this connective were substantial, and so we dedicate an entire section to it (Section 5.5).

5.4 The 'Pairing' Connectives

In this section we will study the group of six connectives (Figure 5.1(c)) that are *obtainable* from each other.

The computational behaviour of logical conjunction is traditionally associated with a pairing operation. In the (generic) style of $\mathcal{X}$, the output and input circuits for conjunction are respectively $[P\widehat{\alpha}, Q\widehat{\beta}]\cdot\gamma$ and $y\cdot[\widehat{x}\widehat{z}R]$. The output circuit builds the components of the pair, while the input circuit decomposes it. It is common practice to split the left introduction rule for conjunction into two rules as follows,

$$\frac{A, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta}\,(\wedge L_1) \qquad \frac{B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta}\,(\wedge L_2)$$

From these, one can derive two computational structures that behave like the traditional projection functions *fst* and *snd*. In an $\mathcal{X}$-style calculus, these rules would yield the two input circuits $y\cdot[\widehat{x}R]$ and $y\cdot[\widehat{z}R]$, which when placed in an interaction with the pair (the 'and output' circuit), can be used to 'select' one of its two components. This functionality is seen in the following principal reduction rules (which are derived from the cut-elimination for conjunction).

$$(\wedge_1): \quad ([P\widehat{\alpha}, Q\widehat{\beta}]\cdot\gamma)\widehat{\gamma} \dagger \widehat{y}(y\cdot[\widehat{x}R]) \;\rightarrow\; P\widehat{\alpha} \dagger \widehat{x}R \;\leftarrow\; \gamma, y \textit{ introduced}$$
$$(\wedge_2): \quad ([P\widehat{\alpha}, Q\widehat{\beta}]\cdot\gamma)\widehat{\gamma} \dagger \widehat{y}(y\cdot[\widehat{z}R]) \;\rightarrow\; Q\widehat{\beta} \dagger \widehat{z}R \;\leftarrow\; \gamma, y \textit{ introduced}$$

This 'selection' behaviour can still be achieved using the invertible left introduction rule, except that a pattern-matching approach of decomposing the pair should be adopted. The reduction rules for an $\mathcal{X}$-style calculus built on invertible rules for conjunction are as follows:

$$(\wedge_3): \quad ([P\widehat{\alpha}, Q\widehat{\beta}]\cdot\gamma)\widehat{\gamma} \dagger \widehat{y}(y\cdot[\widehat{x}\widehat{z}R]) \;\rightarrow\; P\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\beta} \dagger \widehat{z}R) \;\leftarrow\; \gamma, y \textit{ introduced}$$
$$(\wedge_4): \quad ([P\widehat{\alpha}, Q\widehat{\beta}]\cdot\gamma)\widehat{\gamma} \dagger \widehat{y}(y\cdot[\widehat{x}\widehat{z}R]) \;\rightarrow\; Q\widehat{\beta} \dagger \widehat{z}(P\widehat{\alpha} \dagger \widehat{x}R) \;\leftarrow\; \gamma, y \textit{ introduced}$$

The difference between the two variants of the rule is whether *preference* is given to 'selecting' the first or the second component of the pair. Kesner and Cerrito studied the above pattern matching interpretation in a single conclusion 'sequent

calculus' [26], but only considered the $(\wedge_3)$ variant of the conjunction rule (using their own syntax, of course).

The group of six connectives in Figure 5.1(c) each have computational behaviour which can, in a sense, be associated with some pairing functionality. The $\mathcal{X}$-style term constructors that can be extracted from each connective, together with the right-hand sides of the principal reduction rule are shown in Table 5.1.

First notice the similarities between each of the 'pairing' connectives: for each connective, one of the circuits has two blocks, representing the pair itself, while the opposite circuit has a single block. The circuit with the single block provides the functionality to 'select' the components of that pair, and in each case, that circuit can be split into two components (which would be derived from a 'projection' style inference rule). Consider the circuits for the projection style representation of implication. The (generic) implication input circuit $y{\cdot}[P\widehat{\beta}, \widehat{z}Q]$ can be seen as a pair of two terms P and Q, the first binds and output the second binds an input. The output circuits $[\widehat{x}R]{\cdot}\gamma$ and $[R\widehat{\alpha}]{\cdot}\gamma$ can be seen as 'selectors' for the components of the pair created by the input circuit, in much the same way as the *fst* and *snd* operators work on the traditional representation of pairs built using conjunction. Observe the pair of principal reduction rules for this style of representing the circuit,

$$
\begin{array}{ll}
(\rightarrow_1): & ([\widehat{x}R]{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{x}(y{\cdot}[P\widehat{\beta}, \widehat{z}Q]) \rightarrow P\widehat{\beta} \dagger \widehat{x}R \leftarrow \gamma, y \textit{ introduced} \\
(\rightarrow_2): & ([R\widehat{\alpha}]{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{x}(y{\cdot}[P\widehat{\beta}, \widehat{z}Q]) \rightarrow R\widehat{\alpha} \dagger \widehat{z}Q \leftarrow \gamma, y \textit{ introduced}
\end{array}
$$

Some of the similarities between these connectives can be explained if one considers the truth table definitions of the connectives. The six connectives have a feature in common: the truth-value of each connective can be determined (in certain cases) without knowing the value of both of its arguments; we say such connectives can be 'shortcut'. In Figure 5.2, we give the truth table definitions for the six connectives, and in each case give the cases when a connective can be shortcut.

In Section 6.1.2, we will describe a mechanical algorithm for deriving sequent calculus style inference rules from a truth function. We will show the single block circuit for each pairing connective is a direct result of the possibility to shortcut the connective.

	Input Circuit	Output Circuit	'Pattern Matching' RHS
$\wedge$	$y{\cdot}[\widehat{x}\widehat{z}R]$ or $\left\{\begin{array}{l} y{\cdot}[\widehat{x}R] \\ y{\cdot}[\widehat{z}R] \end{array}\right\}$	$[P\widehat{\alpha}, Q\widehat{\beta}]{\cdot}\gamma$	$P\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\beta} \dagger \widehat{z}R)$ or $Q\widehat{\beta} \dagger \widehat{z}(P\widehat{\alpha} \dagger \widehat{x}R)$
$\vee$	$y{\cdot}[\widehat{x}P, \widehat{z}Q]$	$[R\widehat{\alpha}\widehat{\mu}]{\cdot}\gamma$ or $\left\{\begin{array}{l} [R\widehat{\alpha}]{\cdot}\gamma \\ [R\widehat{\mu}]{\cdot}\gamma \end{array}\right\}$	$(R\widehat{\alpha} \dagger \widehat{x}P)\widehat{\mu} \dagger \widehat{z}Q$ or $(R\widehat{\mu} \dagger \widehat{z}Q)\widehat{\alpha} \dagger \widehat{x}P$
$\rightarrow$	$y{\cdot}[P\widehat{\beta}, \widehat{z}Q]$	$[\widehat{x}R\widehat{\alpha}]{\cdot}\gamma$ or $\left\{\begin{array}{l} [\widehat{x}R]{\cdot}\gamma \\ [R\widehat{\alpha}]{\cdot}\gamma \end{array}\right\}$	$(P\widehat{\beta} \dagger \widehat{x}R)\widehat{\alpha} \dagger \widehat{z}Q$ or $P\widehat{\beta} \dagger \widehat{x}(R\widehat{\alpha} \dagger \widehat{z}Q)$
$-$	$y{\cdot}[\widehat{x}R\widehat{\alpha}]$ or $\left\{\begin{array}{l} y{\cdot}[\widehat{x}R] \\ y{\cdot}[R\widehat{\alpha}] \end{array}\right\}$	$[P\widehat{\beta}, \widehat{z}Q]{\cdot}\gamma$	
$\uparrow$	$y{\cdot}[P\widehat{\beta}, Q\widehat{\mu}]$	$[\widehat{x}\widehat{z}R]{\cdot}\gamma$ or $\left\{\begin{array}{l} [\widehat{x}R]{\cdot}\gamma \\ [\widehat{z}R]{\cdot}\gamma \end{array}\right\}$	$P\widehat{\beta} \dagger \widehat{x}(Q\widehat{\mu} \dagger \widehat{z}R)$ or $Q\widehat{\mu} \dagger \widehat{z}(P\widehat{\beta} \dagger \widehat{x}R)$
$\downarrow$	$y{\cdot}[R\widehat{\alpha}\widehat{\beta}]$ or $\left\{\begin{array}{l} y{\cdot}[R\widehat{\alpha}] \\ y{\cdot}[R\widehat{\beta}] \end{array}\right\}$	$[x\widehat{P}, \widehat{z}Q]{\cdot}\gamma$	$(R\widehat{\alpha} \dagger \widehat{x}P)\widehat{\beta} \dagger \widehat{z}Q$ or $(R\widehat{\beta} \dagger \widehat{z}Q)\widehat{\alpha} \dagger \widehat{x}P$

Table 5.1: Circuits and Reduction Rules for the Six 'Pairing' Connectives

A	B	$A\wedge B$	$A\vee B$	$A\rightarrow B$	$A-B$	$A\uparrow B$	$A\downarrow B$
0	0	0	0	1	0	0	0
0	1	0	1	1	1	1	0
1	0	0	1	0	0	1	0
1	1	1	1	1	0	1	1
shortcuts		A=0 or B=0	A=1 or B=1	A=0 or B=1	A=1 or B=0	A=1 or B=1	A=0 or B=0

Figure 5.2: Truth Tables and 'Shortcuts' for the Six 'pairing' Connectives

Definition 5.4.1 ($\mathcal{X}^{\uparrow}$-Syntax) *The circuits of the $\mathcal{X}^{\uparrow}$-calculus are defined by the following grammar, where x, y range over the infinite set of* sockets*, and α, β over* plugs*.*

$$\underset{\textit{capsule}}{\langle x{\cdot}\alpha\rangle} \;\mid\; \underset{\textit{nand output}}{x{\cdot}[P\widehat{\alpha}, Q\widehat{\beta}]} \;\mid\; \underset{\textit{nand input}}{[\widehat{x}\widehat{y}P]{\cdot}\alpha} \;\mid\; \underset{\textit{cut}}{P\widehat{\alpha} \dagger \widehat{x}Q}$$

Definition 5.4.2 (Typing Rules for $X^{\uparrow}$) *The axiom and cut are typed as usual (Definition 5.1.3). The 'nand' input and output circuits are typed as follows.*

$$\frac{P :\cdot \Gamma \vdash \Delta, \alpha{:}A \quad Q :\cdot \Gamma \vdash \Delta, \beta{:}B}{x{\cdot}[P\widehat{\alpha}, Q\widehat{\beta}] :\cdot x{:}A \uparrow B, \Gamma \vdash \Delta}\,(\uparrow L) \qquad \frac{P :\cdot x{:}A, y{:}B, \Gamma \vdash \Delta}{[\widehat{x}\widehat{y}P]{\cdot}\alpha :\cdot \Gamma \vdash \Delta, \alpha{:}A \uparrow B}\,(\uparrow R)$$

Definition 5.4.3 ($\mathcal{X}^{\uparrow}$ Reduction Rules) *We extend the set of* basic reduction rules*, $\mathcal{R}$, (Definition 5.2.4), with the following reduction rules.*

***Left Propagation Rules** : $(\uparrow O\text{-}outs\not\prec)$, $(\uparrow O\text{-}ins\not\prec)$ and $(\uparrow I\not\prec)$*

$$\begin{array}{lll} ([\widehat{x}\widehat{z}P]{\cdot}\gamma)\widehat{\gamma} \not\prec \widehat{y}R & \to ([\widehat{x}\widehat{z}(P\widehat{\gamma} \not\prec \widehat{y}R)]{\cdot}\delta)\widehat{\delta} \dagger \widehat{y}R & \\ ([\widehat{x}\widehat{z}P]{\cdot}\alpha)\widehat{\gamma} \not\prec \widehat{y}R & \to [\widehat{x}\widehat{z}(P\widehat{\gamma} \not\prec \widehat{y}R)]{\cdot}\alpha & \leftarrow\ \gamma{\neq}\alpha \\ (x{\cdot}[P\widehat{\alpha}, Q\widehat{\beta}])\widehat{\gamma} \not\prec \widehat{y}R & \to x{\cdot}[(P\widehat{\gamma} \not\prec \widehat{y}R)\widehat{\alpha}, (Q\widehat{\gamma} \not\prec \widehat{y}R)\widehat{\beta}] & \end{array}$$

***Right Propagation Rules** : $(\nwarrow\!\uparrow I\text{-}outs)$, $(\nwarrow\!\uparrow I\text{-}ins)$ and $(\nwarrow\!\uparrow O)$*

$$\begin{array}{lll} R\widehat{\gamma} \nwarrow \widehat{y}(y{\cdot}[P\widehat{\alpha}, Q\widehat{\beta}]) & \to R\widehat{\gamma} \dagger \widehat{k}(k{\cdot}[(R\widehat{\gamma} \nwarrow \widehat{y}P)\widehat{\alpha}, (R\widehat{\gamma} \nwarrow \widehat{y}Q)\widehat{\beta}]) & \\ R\widehat{\gamma} \nwarrow \widehat{y}(x{\cdot}[P\widehat{\alpha}, Q\widehat{\beta}]) & \to x{\cdot}[(R\widehat{\gamma} \nwarrow \widehat{y}P)\widehat{\alpha}, (R\widehat{\gamma} \nwarrow \widehat{y}Q)\widehat{\beta}]) & \leftarrow\ x{\neq}y \\ R\widehat{\gamma} \nwarrow \widehat{y}([\widehat{x}\widehat{z}P]{\cdot}\alpha) & \to [\widehat{x}\widehat{z}(R\widehat{\gamma} \nwarrow \widehat{y}P)]{\cdot}\alpha) & \end{array}$$

***Renaming Rules** : $(\uparrow I\text{-}rn)$ and $(\uparrow O\text{-}rn)$,*

$$\begin{array}{ll} ([\widehat{x}\widehat{z}P]{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{y}\langle y{\cdot}\alpha\rangle & \to [\widehat{x}\widehat{z}P]{\cdot}\alpha \leftarrow\ \gamma, y \textit{ introduced} \\ \langle z{\cdot}\gamma\rangle\widehat{\gamma} \dagger \widehat{y}(y{\cdot}[P\widehat{\alpha}, Q\widehat{\beta}]) & \to z{\cdot}[P\widehat{\alpha}, Q\widehat{\beta}] \leftarrow\ \gamma, y \textit{ introduced} \end{array}$$

***Principal Reduction Rules** : $(\uparrow_1)$ and $(\uparrow_2)$,*

$$\begin{array}{ll} ([\widehat{x}\widehat{z}R]{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{y}(y{\cdot}[P\widehat{\alpha}, Q\widehat{\beta}]) & \to P\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\beta} \dagger \widehat{z}R) \leftarrow\ \gamma, y \textit{ introduced} \\ ([\widehat{x}\widehat{z}R]{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{y}(y{\cdot}[P\widehat{\alpha}, Q\widehat{\beta}]) & \to Q\widehat{\beta} \dagger \widehat{z}(P\widehat{\alpha} \dagger \widehat{x}R) \leftarrow\ \gamma, y \textit{ introduced} \end{array}$$

Figure 5.3: The $\mathcal{X}^{\uparrow}$-Calculus

5.4.1 Simulations of $\mathcal{X}$

As an experiment, we sought to encode the $\mathcal{X}$-calculus into a calculus which could logically express implication, with the aim of studying simulations.

In this section, we will introduce two target calculi for simulating $\mathcal{X}$. The first is perhaps an obvious choice: the calculus $\mathcal{X}^{\uparrow}$ is based on the functionally complete 'nand' connective. The second, which we call the $\mathcal{X}^{\neg\vee}$-calculus is based on negation and disjunction. Our goal is to see whether each of these calculi (which employ sets of functionally complete connectives as primitives) can encode the syntax of the $\mathcal{X}$-calculus in such a way that reductions are preserved. In the positive case, we shall say the former calculus can *computationally express* the latter.

The $\mathcal{X}^{\uparrow}$-calculus

We give the full Definition of the $\mathcal{X}^{\uparrow}$ calculus in Figure 5.3 following the (mechanical) procedure for deriving Curry-Howard pairs of calculi detailed in Section 5.2.3. Using these definition, we can seek an encoding of $\mathcal{X}$ into $\mathcal{X}^{\uparrow}$ using the logical equivalence $A \rightarrow B \equiv A \uparrow (A \uparrow B)$. We first construct two (partial) derivations that introduce the formula $A \uparrow (A \uparrow B)$ on the left and right hand sides of a sequent, i.e.,

$$\dfrac{\dfrac{\boxed{}}{\Gamma \vdash \Delta, A} \quad \dfrac{\boxed{}}{\dfrac{A, B, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, A \uparrow B}\,(\uparrow R)}}{A \uparrow (A \uparrow B), \Gamma \vdash \Delta}\,(\uparrow L) \qquad \dfrac{\dfrac{\dfrac{}{\boldsymbol{A}, \Gamma \vdash \Delta, A}\,(Ax) \quad \boxed{\phantom{\boldsymbol{A}, \Gamma \vdash \Delta, B}}}{\boldsymbol{A}, A \uparrow B, \Gamma \vdash \Delta}\,(\uparrow L)}{\Gamma \vdash \Delta, A \uparrow (A \uparrow B)}\,(\uparrow R)$$

From the structure of these derivations, we can extract an interpretation of the $\mathcal{X}$-calculus import and export circuits in $\mathcal{X}^{\uparrow}$: (i) the import is encoded as a 'nand' input circuit whose second sub-circuit is a 'nand' output circuit, and (ii) the export is encoded as a 'nand' output circuit over a 'nand' input circuit whose first sub-circuit is a capsule. Note the implicit contraction (highlighted) in the second derivation.

Definition 5.4.4 (Interpretation of $\mathcal{X}$ into $\mathcal{X}^{\uparrow}$)

$$\begin{array}{rll}
\overline{\langle x\cdot\alpha\rangle}^{\uparrow} & = \langle x\cdot\alpha\rangle & \\
\overline{P\widehat{\alpha}\dagger\widehat{x}Q}^{\uparrow} & = \overline{P}^{\uparrow}\widehat{\alpha}\dagger\widehat{x}\overline{Q}^{\uparrow} & \\
\overline{\widehat{x}P\widehat{\alpha}\cdot\gamma}^{\uparrow} & = [\widehat{x}\widehat{z}(z\cdot[\langle x\cdot\pi\rangle\widehat{\pi},\overline{P}^{\uparrow}\widehat{\alpha}])]\cdot\gamma & z,\pi \textit{ fresh} \\
\overline{P\widehat{\alpha}\,[y]\,\widehat{x}Q}^{\uparrow} & = y\cdot[\overline{P}^{\uparrow}\widehat{\alpha},([\widehat{z}\widehat{x}\overline{Q}^{\uparrow}]\cdot\pi)\widehat{\pi}] & z,\pi \textit{ fresh}
\end{array}$$

Now we must check that reductions of $\mathcal{X}$ (Definitions 3.1.4 and 3.1.7) can be simulated by those of $\mathcal{X}^{\uparrow}$ (Definition 5.4.3). As discussed in Section 5.2, the propagation and renaming rules are generic to any $\mathcal{X}$-style term calculus and perform the same basic task of (i) pushing cuts through the structure of sub-circuits and, (ii) renaming an outermost free connector. We therefore only concern ourselves with the simulation of the $\mathcal{X}$-calculus rules $(\textit{exp-imp}_{cbn})$ and $(\textit{exp-imp}_{cbv})$. We also note that $\mathcal{X}^{\uparrow}$ can be extended with generalised rules for garbage collection and renaming (c.f., Lemmas 3.1.21 and 3.1.25 respectively).

Lemma 5.4.5 (Simulation of $\mathcal{X}$ in $\mathcal{X}^{\uparrow}$) *Recall the principal reduction rules of the $\mathcal{X}$-calculus (built on the implication connective) are,*

$$\begin{array}{l}
(\textit{exp-imp}_{cbv}) : (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R) \;\rightarrow\; Q\widehat{\gamma}\dagger\widehat{y}(P\widehat{\beta}\dagger\widehat{z}R) \;\leftarrow\; \alpha, x \textit{ introduced} \\
(\textit{exp-imp}_{cbn}) : (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R) \;\rightarrow\; (Q\widehat{\gamma}\dagger\widehat{y}P)\widehat{\beta}\dagger\widehat{z}R \;\leftarrow\; \alpha, x \textit{ introduced}
\end{array}$$

Interpreting the left-hand side of the rules,

$$\begin{array}{ll}
 & \overline{(\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R)}^{\uparrow} \\
= & \overline{(\widehat{y}P\widehat{\beta}\cdot\alpha)}^{\uparrow}\widehat{\alpha}\dagger\widehat{x}\overline{(Q\widehat{\gamma}\,[x]\,\widehat{z}R)}^{\uparrow} \\
= & ([\widehat{y}\widehat{v}(v\cdot[\langle y\cdot\pi\rangle\widehat{\pi},\overline{P}^{\uparrow}\widehat{\beta}])]\cdot\alpha)\widehat{\boldsymbol{\alpha}}\dagger\widehat{\boldsymbol{x}}(x\cdot[\overline{Q}^{\uparrow}\widehat{\gamma},([\widehat{w}\widehat{z}\overline{R}^{\uparrow}]\cdot\sigma)\widehat{\sigma}]) \qquad (5.1)
\end{array}$$

Where π,σ,w,v are fresh in Circuit (5.1)*.*

Applying the first variant of the principal reduction rule for $\mathcal{X}^{\uparrow}$ called $(\uparrow_1)$ from Definition 5.4.3 to the cut $\widehat{\alpha}\dagger\widehat{x}$ (highlighted above), we get:

$$\begin{array}{ll}
\overline{Q}^{\uparrow}\widehat{\gamma}\dagger\widehat{y}(([\widehat{w}\widehat{z}\overline{R}^{\uparrow}]\cdot\sigma)\widehat{\sigma}\dagger\widehat{\boldsymbol{v}}(v\cdot[\langle y\cdot\pi\rangle\widehat{\pi},\overline{P}^{\uparrow}\widehat{\beta}])) & \rightarrow (\uparrow_1) \\
\overline{Q}^{\uparrow}\widehat{\gamma}\dagger\widehat{y}(\langle y\cdot\pi\rangle\widehat{\boldsymbol{\pi}}\dagger\widehat{\boldsymbol{w}}(\overline{P}^{\uparrow}\widehat{\beta}\dagger\widehat{z}\overline{R}^{\uparrow})) & \rightarrow (\textit{ren}\text{-R}) \\
\overline{Q}^{\uparrow}\widehat{\gamma}\dagger\widehat{y}(\overline{P}^{\uparrow}\widehat{\beta}\dagger\widehat{z}\overline{R}^{\uparrow}) & = \\
\overline{Q\widehat{\gamma}\dagger\widehat{y}(P\widehat{\beta}\dagger\widehat{z}R)}^{\uparrow} &
\end{array}$$

The above is the interpretation of the right-hand side of the $(\textit{exp-imp}_{cbv})$ *rule in* $\mathcal{X}^{\uparrow}$.

We cannot exactly simulate the other rule $(\textit{exp-imp}_{cbn})$ *in* $\mathcal{X}^{\uparrow}$. *Instead, by first applying the second variant of the principal reduction rule for* $\mathcal{X}^{\uparrow}$ *called* $(\uparrow_2)$, *to Circuit* (5.1), *we get:*

$$\begin{array}{lll}
([\widehat{w}\widehat{z}\overline{R}^{\uparrow}]\cdot\sigma)\widehat{\sigma}\dagger\widehat{v}(\overline{Q}^{\uparrow}\widehat{\gamma}\dagger\widehat{\boldsymbol{y}}(v\cdot[\langle y\cdot\pi\rangle\widehat{\pi},\overline{P}^{\uparrow}\widehat{\beta}])) & \rightarrow & \\
([\widehat{w}\widehat{z}\overline{R}^{\uparrow}]\cdot\sigma)\widehat{\sigma}\dagger\widehat{v}(v\cdot[(\overline{Q}^{\uparrow}\widehat{\gamma}\dagger\widehat{y}\langle y\cdot\pi\rangle)\widehat{\pi},(\overline{Q}^{\uparrow}\widehat{\gamma}\nwarrow\widehat{y}\overline{P}^{\uparrow})\widehat{\beta}]) & \rightarrow & (\textit{ren}\text{-L}) \\
([\widehat{w}\widehat{z}\overline{R}^{\uparrow}]\cdot\sigma)\widehat{\sigma}\dagger\widehat{v}(v\cdot[\overline{Q\{\pi/\gamma\}}^{\uparrow}\widehat{\pi},(\overline{Q}^{\uparrow}\widehat{\gamma}\nwarrow\widehat{y}\overline{P}^{\uparrow})\widehat{\beta}]) & =_{\alpha} & \\
([\widehat{w}\widehat{z}\overline{R}^{\uparrow}]\cdot\sigma)\widehat{\sigma}\dagger\widehat{v}(v\cdot[\overline{Q}^{\uparrow}\widehat{\gamma},(\overline{Q}^{\uparrow}\widehat{\gamma}\nwarrow\widehat{y}\overline{P}^{\uparrow})\widehat{\beta}]) &
\end{array}$$

Again we have a choice of whether to apply $(\uparrow_1)$ *or* $(\uparrow_2)$.

(i) Applying $(\uparrow_1)$, *we get:*

$$\overline{Q}^{\uparrow}\widehat{\gamma}\dagger\widehat{w}((\overline{Q}^{\uparrow}\widehat{\gamma}\nwarrow\widehat{y}\overline{P}^{\uparrow})\widehat{\beta}\dagger\widehat{z}\overline{R}^{\uparrow}) \tag{5.2}$$

Which reduces by $(\textit{act}\text{-R})$ *and* $(\nwarrow gc)$ *to,*

$$(\overline{Q}^{\uparrow}\widehat{\gamma}\nwarrow\widehat{y}\overline{P}^{\uparrow})\widehat{\beta}\dagger\widehat{z}\overline{R}^{\uparrow}$$

This almost *matches the right-hand side of* $(\textit{exp-imp}_{cbn})$, *but in general, we cannot deactivate the right-propagating cut at this point since* $\overline{P}^{\uparrow}$ *may not introduce* y.

Also, in Circuit (5.2), *since* w *was freshly introduced by the interpretation, it can reduce to* $\overline{Q}^{\uparrow}$ *in the case where* $\overline{Q}^{\uparrow}$ *does not introduce* γ.

(ii) The alternative choice is to apply $(\uparrow_2)$, *giving:*

$$\begin{array}{l}
(\overline{Q}^{\uparrow}\widehat{\gamma}\nwarrow\widehat{y}\overline{P}^{\uparrow})\widehat{\beta}\dagger\widehat{z}(\overline{Q}^{\uparrow}\widehat{\gamma}\dagger\widehat{w}\overline{R}^{\uparrow}) \rightarrow (\textit{act}\text{-R}),(\nwarrow gc) \\
(\overline{Q}^{\uparrow}\widehat{\gamma}\nwarrow\widehat{y}\overline{P}^{\uparrow})\widehat{\beta}\dagger\widehat{z}\overline{R}^{\uparrow}
\end{array}$$

Which again features the same problem as part (i). We also note that where $\overline{Q}^{\uparrow}$ *does not introduce* γ, *the reduct* $(\overline{Q}^{\uparrow}\widehat{\gamma}\nwarrow\widehat{y}\overline{P}^{\uparrow})\widehat{\beta}\dagger\widehat{z}\overline{Q}^{\uparrow}$ *is also obtainable.*

The problems encountered simulating $\mathcal{X}$ in $\mathcal{X}^{\uparrow}$ were unexpected. In fact, we were able to show that an encoding of $\mathcal{X}$ into a $\mathcal{X}^{\downarrow}$-calculus (a Curry-Howard calculus in the style of $\mathcal{X}$ that employed the nor connective as a primitive) could only fully simulate the $(\textit{exp-imp}_{cbn})$ variant of the principal reduction rule for $\mathcal{X}$.

With this negative result, we sought a calculus which might be able to fully simulate $\mathcal{X}$: a 'computational equivalent' of $\mathcal{X}$. We built a Curry-Howard pair of calculi based on the functionally complete set of connectives $\{\neg,\vee\}$.

The Calculus, $\mathcal{X}^{\neg\vee}$

The full definition of the $\mathcal{X}^{\neg\vee}$-calculus is given in Figure 5.4. Examining truth tables, we find that the formula $\neg A \vee B$ is logically equivalent $A \rightarrow B$.

Following the same procedure we did to interpret $\mathcal{X}$ into $\mathcal{X}^{\uparrow}$, we build (partial) derivations of $\Gamma \vdash \Delta, \neg A \vee B$ and $\neg A \vee B, \Gamma \vdash \Delta$. These are shown below.

$$\dfrac{\dfrac{\dfrac{\quad}{A, \Gamma \vdash \Delta, B}}{\Gamma \vdash \Delta, \neg A, B}\,(\neg R)}{\Gamma \vdash \Delta, \neg A \vee B}\,(\vee R) \qquad \dfrac{\dfrac{\dfrac{\quad}{\Gamma \vdash \Delta, A}}{\neg A, \Gamma \vdash \Delta}\,(\neg L) \qquad \dfrac{\quad}{\Gamma \vdash \Delta, B}}{\neg A \vee B, \Gamma \vdash \Delta}\,(\vee L)$$

Notice that in this case, the larger sets of connectives leads to simpler encodings compared to the $\mathcal{X}^{\uparrow}$-calculus where only a single connective was considered. We give the corresponding interpretation of $\mathcal{X}$ into $\mathcal{X}^{\neg\vee}$ below.

Definition 5.4.6 (Interpretation of $\mathcal{X}$ into $\mathcal{X}^{\neg\vee}$)

$$\begin{aligned}
\overline{\langle x \cdot \alpha \rangle}^{\vee\neg} &= \langle x \cdot \alpha \rangle \\
\overline{P\widehat{\alpha} \dagger \widehat{x}Q}^{\vee\neg} &= \overline{P}^{\vee\neg}\widehat{\alpha} \dagger \widehat{x}\overline{Q}^{\vee\neg} \\
\overline{\widehat{x}P\widehat{\alpha} \cdot \gamma}^{\vee\neg} &= [([\widehat{x}\overline{P}^{\vee\neg}] \cdot \beta)\widehat{\beta}\widehat{\alpha}] \cdot \gamma \qquad \beta \textit{ fresh} \\
\overline{P\widehat{\alpha}\,[y]\,\widehat{x}Q}^{\vee\neg} &= y \cdot [\widehat{z}(z \cdot [\overline{P}^{\vee\neg}\widehat{\alpha}]), \widehat{x}Q] \quad z \textit{ fresh}
\end{aligned}$$

Now we must check $\mathcal{X}^{\neg\vee}$ can simulate $\mathcal{X}$. We again restrict attention to the simulation of the rules $(\textit{exp-imp}_{cbn})$ and $(\textit{exp-imp}_{cbv})$, and make use of generalised garbage collection and renaming rules for convenience.

Lemma 5.4.7 (Simulation of $\mathcal{X}$ in $\mathcal{X}^{\neg\vee}$) *We begin with interpretation of the left-hand of an (exp-imp) rule,*

$$\begin{aligned}
&\overline{(\widehat{y}P\widehat{\beta} \cdot \alpha)\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R)}^{\vee\neg} \\
&= \overline{(\widehat{y}P\widehat{\beta} \cdot \alpha)}^{\vee\neg}\widehat{\alpha} \dagger \widehat{x}\,\overline{(Q\widehat{\gamma}\,[x]\,\widehat{z}R)}^{\vee\neg} \\
&= ([([\widehat{y}\overline{P}^{\vee\neg}] \cdot \mu)\widehat{\mu}\widehat{\beta}] \cdot \alpha)\widehat{\alpha} \dagger \widehat{x}(x \cdot [\widehat{w}(w \cdot [\overline{Q}^{\vee\neg}\widehat{\gamma}]), \widehat{z}\overline{R}^{\vee\neg}]) \qquad (5.3)
\end{aligned}$$

There are two cases to consider: (i) reducing the cut $\widehat{\alpha} \dagger \widehat{x}$ by $(\vee_1)$ or, (ii) by $(\vee_2)$,

(i) Applying $(\vee_1)$, we obtain,

$$(([\widehat{y}\overline{P}^{\vee\neg}] \cdot \gamma)\widehat{\gamma} \dagger \widehat{w}([\widehat{w}\overline{Q}^{\vee\neg}] \cdot \gamma))\widehat{\beta} \dagger \widehat{z}\overline{R}^{\vee\neg} \rightarrow (\neg) \qquad (5.4)$$

$$\begin{array}{ll} (\overline{Q}^{\vee\neg}\widehat{\gamma} \dagger \widehat{y}\overline{P}^{\vee\neg})\widehat{\beta} \dagger \widehat{z}\overline{R}^{\vee\neg} & = \\ \overline{(Q\widehat{\gamma} \dagger \widehat{y}P)\widehat{\beta} \dagger \widehat{z}R}^{\vee\neg} & = \end{array}$$

Where u, v, s, t are fresh.

Which is the interpretation of the right-hand side of $(\textit{exp-imp}_{cbn})$.

(ii) Applying $(\vee_2)$ to Circuit (5.3)*, we obtain,*

$$(([\widehat{y}\overline{P}^{\vee\neg}]\cdot\mu)\widehat{\beta} \dagger \widehat{z}\overline{R}^{\vee\neg})\widehat{\mu} \dagger \widehat{w}(w\cdot[\overline{Q}^{\vee\neg}\widehat{\gamma}])$$

If we evaluate the outermost cut by applying the rules $(\textit{act-}\textsc{l})$, $(\textit{cut}\not\prec)$, $(\not\prec \textit{gc})$, $(\not\prec d)$, we get back to Circuit (5.4)*.*

Instead, we propagate the innermost cut through the structure of the not output circuit by applying the rules $(\textit{act-}\textsc{l})$ then $(\neg\textit{O-ins}\not\prec)$, giving:

$$\begin{array}{l} ([\widehat{y}(\overline{P}^{\vee\neg}\widehat{\beta} \not\prec \widehat{z}\overline{R}^{\vee\neg})]\cdot\mu)\widehat{\mu} \dagger \widehat{w}(w\cdot[\overline{Q}^{\vee\neg}\widehat{\gamma}]) \rightarrow (\neg) \\ \overline{Q}^{\vee\neg}\widehat{\gamma} \dagger \widehat{y}(\overline{P}^{\vee\neg}\widehat{\beta} \not\prec \widehat{z}\overline{R}^{\vee\neg}) \end{array}$$

Notice that this is almost *the interpretation of the right-hand side of the rule $(\textit{exp-imp}_{cbv})$, except the innermost cut is activated to the left.*

The negative results in simulating the $\mathcal{X}$-calculus suggests a bias can be introduced to computational calculi depending on the choice of logical connective. It would be interesting to determine the exact cause of the bias.

In Table 5.1), notice how the permutations of each of the circuits in the 'Pattern Matching RHS' differ from each other. The implication and difference connectives stand out from the other connectives in that they share the same right-hand sides. Since the single block of the pair 'selecting' circuit for these two connectives bind an input and an output, as opposed to two connectors of the same kind, the permutation of right-hand sides differ only on how the cuts are bracketed.

Our first intuitions led us to believe that the way in which the rule's right-hand side was bracketed was the cause of the bias: notice in the simulation results, the target calculus could only fully simulate the variant of $\mathcal{X}$'s principal reduction rule which it was bracketed towards. For example, the right-hand sides of the 'nand' rule are bracketed to the right, and the encoding of $\mathcal{X}$ into $\mathcal{X}^{\uparrow}$ could only fully simulate the rule $(\textit{exp-imp}_{cbv})$ which is also bracketed to the right.

Since the right-hand side of the rules $(\vee_1)$ and $(\vee_2)$ are bracketed to the left and the rule $(\uparrow_1)$ and $(\uparrow_2)$ are bracketed to the right, we sought to simulate the dis-

junctive fragment of $\mathcal{X}^{\neg\vee}$ (i.e., $\mathcal{X}^{\vee}$) in $\mathcal{X}^{\uparrow}$. Had our intuition been correct a simulation would not have been possible. Fortunately, the intuition was incorrect. In the following, we will give an encoding of $\mathcal{X}^{\vee}$ into $\mathcal{X}^{\uparrow}$ and show that we can simulate the rule $(\vee_1)$. We will also show that with a second (different) encoding, we can simulate the rule $(\vee_2)$.

Encoding $\mathcal{X}^{\vee}$ into $\mathcal{X}^{\uparrow}$

We will consider the disjunctive-fragment of the $\mathcal{X}^{\neg\vee}$-calculus given in Figure 5.4. Using the equivalence $A \vee B \equiv (A \uparrow A) \uparrow (B \uparrow B)$, we can build the following derivations of $(A \uparrow A) \uparrow (B \uparrow B), \Gamma \vdash \Delta$ and $\Gamma \vdash \Delta, (A \uparrow A) \uparrow (B \uparrow B)$:

$$\dfrac{\dfrac{\dfrac{\boxed{}}{A, \Gamma \vdash \Delta}}{\Gamma \vdash \Delta, A \uparrow A}(\uparrow R) \qquad \dfrac{\dfrac{\boxed{}}{B, B, \Gamma \vdash \Delta}}{\Gamma \vdash \Delta, B \uparrow B}(\uparrow R)}{(A \uparrow A) \uparrow (B \uparrow B), \Gamma \vdash \Delta}(\uparrow L)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\boxed{}\quad\boxed{}}{\Gamma \vdash \Delta, A, B \quad \Gamma \vdash \Delta, A, B}}{A \uparrow A, \Gamma \vdash \Delta, B}(\uparrow L) \qquad \dfrac{\dfrac{\boxed{}\quad\boxed{}}{\Gamma \vdash \Delta, A, B \quad \Gamma \vdash \Delta, A, B}}{A \uparrow A, \Gamma \vdash \Delta, B}(\uparrow L)}{A \uparrow A, B \uparrow B, \Gamma \vdash \Delta}(\uparrow L)}{\Gamma \vdash \Delta, (A \uparrow A) \uparrow (B \uparrow B)}(\uparrow R)$$

This gives the following interpretation of $\mathcal{X}^{\vee}$ into $\mathcal{X}^{\uparrow}$.

Definition 5.4.11 (Interpretation of $\mathcal{X}^{\vee}$ into $\mathcal{X}^{\uparrow}$)

$$\begin{aligned}
\overline{\langle x \cdot \alpha \rangle} &= \langle x \cdot \alpha \rangle \\
\overline{P\widehat{\alpha} \dagger \widehat{x}Q} &= \overline{P}\widehat{\alpha} \dagger \widehat{x}\overline{Q} \\
\overline{z \cdot [\widehat{x}P, \widehat{y}Q]} &= z \cdot [([\widehat{x}\widehat{u}\overline{P}] \cdot \alpha)\widehat{\alpha}, ([\widehat{y}\widehat{v}\overline{Q}] \cdot \beta)\widehat{\beta}] \\
\overline{[R\widehat{\alpha}\widehat{\beta}] \cdot \gamma} &= [\widehat{u}\widehat{v}(v \cdot [(u \cdot [\overline{R}\widehat{\alpha}, \overline{R}\widehat{\alpha}])\widehat{\beta}, (u \cdot [\overline{R}\widehat{\alpha}, \overline{R}\widehat{\alpha}])\widehat{\beta}])] \cdot \gamma
\end{aligned}$$

Where u, v, α, β are fresh.

The interpretation of the 'or output' circuit is unusual: notice the duplication of the sub-circuit R four times, and the contraction of the socket u. Again, allowing generalised renaming and garbage collection rules, we attempt to simulate $\mathcal{X}^{\vee}$ in $\mathcal{X}^{\uparrow}$.

Definition 5.4.8 ($\mathcal{X}^{\neg\vee}$-Syntax) *The circuits of $\mathcal{X}^{\neg\vee}$ are defined by the following grammar, where x, y, z range over the infinite set of* sockets*, and α, β, γ over* plugs*.*

$$P, Q ::= \underset{\textit{capsule}}{\langle x{\cdot}\alpha\rangle} \mid \underset{\textit{not input}}{x{\cdot}[P\widehat{\alpha}]} \mid \underset{\textit{not output}}{[\widehat{x}P]{\cdot}\alpha} \mid \underset{\textit{or input}}{z{\cdot}[\widehat{x}P, \widehat{y}Q]} \mid \underset{\textit{or output}}{[P\widehat{\alpha}\widehat{\beta}]{\cdot}\gamma} \mid \underset{\textit{cut}}{P\widehat{\alpha} \dagger \widehat{x}Q}$$

Definition 5.4.9 (Typing Rules for $\mathcal{X}^{\neg\vee}$) *The axiom and cut are typed as usual (Definition 5.1.3). The input and output circuits for 'not' and 'and' are typed as follows.*

$$\frac{P :\cdot \Gamma \vdash \Delta, \alpha{:}A}{x{\cdot}[P\widehat{\alpha}] :\cdot x{:}\neg A, \Gamma \vdash \Delta}\,(\neg L) \qquad \frac{P :\cdot x{:}A, \Gamma \vdash \Delta}{[\widehat{x}P]{\cdot}\alpha :\cdot \Gamma \vdash \Delta, \alpha{:}\neg A}\,(\neg R)$$

$$\frac{P :\cdot x{:}A, \Gamma \vdash \Delta \qquad Q :\cdot y{:}B, \Gamma \vdash \Delta}{z{\cdot}[\widehat{x}P, \widehat{y}Q] :\cdot z{:}A \vee B, \Gamma \vdash \Delta}\,(\vee L) \qquad \frac{P :\cdot \Gamma \vdash \Delta, \alpha{:}A, \beta{:}B}{[P\widehat{\alpha}\widehat{\beta}]{\cdot}\gamma :\cdot \Gamma \vdash \Delta, \gamma{:}A \vee B}\,(\vee R)$$

Definition 5.4.10 ($\mathcal{X}^{\neg\vee}$ Reduction Rules) *We extend Definition 5.2.4 with,*

***Left Propagation Rules** :*

$$\begin{array}{rlll}
(\neg O\text{-}outs\not\prec): & ([\widehat{x}P]{\cdot}\gamma)\widehat{\gamma} \not\prec \widehat{y}R & \to ([\widehat{x}(P\widehat{\gamma} \not\prec \widehat{y}R)]{\cdot}\delta)\widehat{\delta} \dagger \widehat{y}R & \\
(\neg O\text{-}ins\not\prec): & ([\widehat{x}P]{\cdot}\alpha)\widehat{\gamma} \not\prec \widehat{y}R & \to [\widehat{x}(P\widehat{\gamma} \not\prec \widehat{y}R)]{\cdot}\alpha & \leftarrow \gamma{\neq}\alpha \\
(\neg I\not\prec): & (x{\cdot}[P\widehat{\alpha}])\widehat{\gamma} \not\prec \widehat{y}R & \to x{\cdot}[(P\widehat{\gamma} \not\prec \widehat{y}R)\widehat{\alpha}] & \\
(\vee O\text{-}outs\not\prec): & ([P\widehat{\alpha}\widehat{\beta}]{\cdot}\gamma)\widehat{\gamma} \not\prec \widehat{y}R & \to ([(P\widehat{\gamma} \not\prec \widehat{y}R)\widehat{\alpha}\widehat{\beta}]{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{y}R & \\
(\vee O\text{-}ins\not\prec): & ([P\widehat{\alpha}\widehat{\beta}]{\cdot}\delta)\widehat{\gamma} \not\prec \widehat{y}R & \to [(P\widehat{\gamma} \not\prec \widehat{y}R)\widehat{\alpha}\widehat{\beta}]{\cdot}\delta & \leftarrow \gamma{\neq}\delta \\
(\vee I\not\prec): & (z{\cdot}[\widehat{x}P, \widehat{w}Q])\widehat{\gamma} \not\prec \widehat{y}R & \to z{\cdot}[\widehat{x}(P\widehat{\gamma} \not\prec \widehat{y}R), \widehat{w}(Q\widehat{\gamma} \not\prec \widehat{y}R)] &
\end{array}$$

***Right Propagation Rules** :*

$$\begin{array}{rlll}
(\not\prec\neg I\text{-}outs): & R\widehat{\gamma} \not\prec \widehat{y}(y{\cdot}[P\widehat{\alpha}]) & \to R\widehat{\gamma} \dagger \widehat{y}(y{\cdot}[(R\widehat{\gamma} \not\prec \widehat{y}P)\widehat{\alpha}]) & \\
(\not\prec\neg I\text{-}ins): & R\widehat{\gamma} \not\prec \widehat{y}(x{\cdot}[P\widehat{\alpha}]) & \to x{\cdot}[(R\widehat{\gamma} \not\prec \widehat{y}P)\widehat{\alpha}] & \leftarrow y{\neq}x \\
(\not\prec\neg O): & R\widehat{\gamma} \not\prec \widehat{y}([\widehat{x}P]{\cdot}\alpha) & \to [\widehat{x}(R\widehat{\gamma} \not\prec \widehat{y}P)]{\cdot}\alpha & \\
(\not\prec\vee I\text{-}outs): & R\widehat{\gamma} \not\prec \widehat{y}(y{\cdot}[\widehat{x}P, \widehat{z}Q]) & \to R\widehat{\gamma} \dagger \widehat{k}(k{\cdot}[\widehat{x}(R\widehat{\gamma} \not\prec \widehat{y}P), \widehat{z}(R\widehat{\gamma} \not\prec \widehat{y}Q)]) & \\
(\not\prec\vee I\text{-}ins): & R\widehat{\gamma} \not\prec \widehat{y}(w{\cdot}[\widehat{x}P, \widehat{z}Q]) & \to w{\cdot}[\widehat{x}(R\widehat{\gamma} \not\prec \widehat{y}P), \widehat{z}(R\widehat{\gamma} \not\prec \widehat{y}Q)] & \leftarrow w{\neq}y \\
(\not\prec\vee O): & R\widehat{\gamma} \not\prec \widehat{y}([P\widehat{\alpha}\widehat{\beta}]{\cdot}\delta) & \to [(R\widehat{\gamma} \not\prec \widehat{y}P)\widehat{\alpha}\widehat{\beta}]{\cdot}\delta &
\end{array}$$

***Renaming Rules** :*

$$\begin{array}{rlll}
(\neg I\text{-}rn): & ([\widehat{x}P]{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{y}\langle y{\cdot}\alpha\rangle & \to [\widehat{x}P]{\cdot}\alpha & \leftarrow \gamma \textit{ introduced} \\
(\neg O\text{-}rn): & \langle z{\cdot}\gamma\rangle\widehat{\gamma} \dagger \widehat{y}(y{\cdot}[P\widehat{\alpha}]) & \to z{\cdot}[P\widehat{\alpha}] & \leftarrow y \textit{ introduced} \\
(\vee I\text{-}rn): & ([R\widehat{\alpha}\widehat{\beta}]{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{y}\langle y{\cdot}\delta\rangle & \to [R\widehat{\alpha}\widehat{\beta}]{\cdot}\delta & \leftarrow \gamma \textit{ introduced} \\
(\vee O\text{-}rn): & \langle w{\cdot}\gamma\rangle\widehat{\gamma} \dagger \widehat{y}(y{\cdot}[\widehat{x}P, \widehat{z}Q]) & \to w{\cdot}[\widehat{x}P, \widehat{z}Q] & \leftarrow y \textit{ introduced}
\end{array}$$

***Principal Reduction Rules** :*

$$\begin{array}{rlll}
(\neg): & ([\widehat{x}Q]{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{y}(y{\cdot}[P\widehat{\beta}]) & \to P\widehat{\beta} \dagger \widehat{x}Q & \leftarrow \gamma, y \textit{ introduced} \\
(\vee_1): & ([R\widehat{\alpha}\widehat{\beta}]{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{y}(y{\cdot}[\widehat{x}P, \widehat{z}Q]) & \to (R\widehat{\alpha} \dagger \widehat{x}P)\widehat{\beta} \dagger \widehat{z}Q & \leftarrow \gamma, y \textit{ introduced} \\
(\vee_2): & ([R\widehat{\alpha}\widehat{\beta}]{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{y}(y{\cdot}[\widehat{x}P, \widehat{z}Q]) & \to (R\widehat{\beta} \dagger \widehat{z}Q)\widehat{\alpha} \dagger \widehat{x}P & \leftarrow \gamma, y \textit{ introduced}
\end{array}$$

Figure 5.4: The $\mathcal{X}^{\neg\vee}$-Calculus

Lemma 5.4.12 (Simulation of $\mathcal{X}^{\vee}$ in $\mathcal{X}^{\uparrow}$) *We begin with interpretation of the left-hand side of a* $(\vee)$ *rule (see Figure 5.4),*

$$\begin{aligned}
&\overline{([R\widehat{\alpha}\widehat{\beta}]\cdot\gamma)\widehat{\gamma}\dagger\widehat{y}(y\cdot[\widehat{x}P,\widehat{z}Q])}\\
=\ &\overline{([R\widehat{\alpha}\widehat{\beta}]\cdot\gamma)\widehat{\gamma}\dagger\widehat{y}(y\cdot[\widehat{x}P,\widehat{z}Q])}\\
=\ &([\widehat{u}\widehat{v}(v\cdot[(u\cdot[\overline{R}\widehat{\alpha},\overline{R}\widehat{\alpha}])\widehat{\beta},(u\cdot[\overline{R}\widehat{\alpha},\overline{R}\widehat{\alpha}])\widehat{\beta}])]\cdot\gamma)\widehat{\gamma}\dagger\widehat{y}(y\cdot[([\widehat{x}\widehat{s}\overline{P}]\cdot\mu)\widehat{\mu},([\widehat{z}\widehat{t}\overline{Q}]\cdot\delta)\widehat{\delta}]) \quad (5.5)
\end{aligned}$$

Where u,v,s,t,δ,μ *are fresh. There are two cases to consider: (i) reducing the cut* $\widehat{\alpha}\dagger\widehat{x}$ *by* $(\uparrow_1)$ *or, (ii) by* $(\uparrow_2)$ *(see Figure 5.3),*

(i) By $(\uparrow_1)$*,*

$$\begin{aligned}
&([\widehat{x}\widehat{s}\overline{P}]\cdot\mu)\widehat{\mu}\dagger\widehat{u}(([\widehat{z}\widehat{t}\overline{Q}]\cdot\delta)\widehat{\delta}\dagger\widehat{v}(v\cdot[(u\cdot[\overline{R}\widehat{\alpha},\overline{R}\widehat{\alpha}])\widehat{\beta},(u\cdot[\overline{R}\widehat{\alpha},\overline{R}\widehat{\alpha}])\widehat{\beta}])) \ \to(\uparrow_1)\\
&([\widehat{x}\widehat{s}\overline{P}]\cdot\mu)\widehat{\mu}\dagger\widehat{u}((u\cdot[\overline{R}\widehat{\alpha},\overline{R}\widehat{\alpha}])\widehat{\beta}\dagger\widehat{z}((u\cdot[\overline{R}\widehat{\alpha},\overline{R}\widehat{\alpha}])\widehat{\beta}\dagger\widehat{t}\overline{Q}))
\end{aligned}$$

Propagating the outermost cut through the structure by applying $(act\text{-}\mathrm{R})$*,* $(\nwarrow cut)$*,* $(\nwarrow\uparrow I\text{-}outs)$*,* $(\nwarrow cut)$*,* $(\nwarrow\uparrow I\text{-}outs)$ *then, applying (Lem. 3.1.21) four times, we get:*

$$\begin{aligned}
&(([\widehat{x}\widehat{s}\overline{P}]\cdot\mu)\widehat{\mu}\dagger\widehat{u}(u\cdot[\overline{R}\widehat{\alpha},\overline{R}\widehat{\alpha}]))\widehat{\beta}\dagger\widehat{z}((([\widehat{x}\widehat{s}\overline{P}]\cdot\mu)\widehat{\mu}\dagger\widehat{u}(u\cdot[\overline{R}\widehat{\alpha},\overline{R}\widehat{\alpha}]))\widehat{\beta}\dagger\widehat{t}\overline{Q}) \ \to(\uparrow_1)\\
&(\overline{R}\widehat{\alpha}\dagger\widehat{x}(\overline{R}\widehat{\alpha}\dagger\widehat{s}\overline{P}))\widehat{\beta}\dagger\widehat{z}((([\widehat{x}\widehat{s}\overline{P}]\cdot\mu)\widehat{\mu}\dagger\widehat{u}(u\cdot[\overline{R}\widehat{\alpha},\overline{R}\widehat{\alpha}]))\widehat{\beta}\dagger\widehat{t}\overline{Q}) \ \to(\uparrow_1)\\
&(\overline{R}\widehat{\alpha}\dagger\widehat{x}(\overline{R}\widehat{\alpha}\dagger\widehat{s}\overline{P}))\widehat{\beta}\dagger\widehat{z}((\overline{R}\widehat{\alpha}\dagger\widehat{x}(\overline{R}\widehat{\alpha}\dagger\widehat{s}\overline{P}))\widehat{\beta}\dagger\widehat{t}\overline{Q})
\end{aligned}$$

Now we will choose *to eliminate the cuts formed with the fresh connectors. Applying* $(act\text{-}\mathrm{R})$ *then* $(\nwarrow gc)$*, we get:*

$$\begin{aligned}
&(\overline{R}\widehat{\alpha}\dagger\widehat{x}\overline{P})\widehat{\beta}\dagger\widehat{z}((\overline{R}\widehat{\alpha}\dagger\widehat{x}(\overline{R}\widehat{\alpha}\dagger\widehat{s}\overline{P}))\widehat{\beta}\dagger\widehat{t}\overline{Q}) \ \to(act\text{-}\mathrm{R}),(\nwarrow gc)\\
&\frac{(\overline{R}\widehat{\alpha}\dagger\widehat{x}\overline{P})\widehat{\beta}\dagger\widehat{z}\overline{Q}}{(R\widehat{\alpha}\dagger\widehat{x}P)\widehat{\beta}\dagger\widehat{z}Q} \qquad =
\end{aligned}$$

$(\vee_1)$ *nor the interpretation of* $(\vee_2)$*.*

(ii) By $(\uparrow_2)$*,*

$$([\widehat{z}\widehat{t}\overline{Q}]\cdot\beta)\widehat{\beta}\dagger\widehat{v}(([\widehat{x}\widehat{s}\overline{P}]\cdot\alpha)\widehat{\alpha}\dagger\widehat{u}(v\cdot[(u\cdot[\overline{R}\widehat{\alpha},\overline{R}\widehat{\alpha}])\widehat{\beta},(u\cdot[\overline{R}\widehat{\alpha},\overline{R}\widehat{\alpha}])\widehat{\beta}]))$$

Reducing the outermost cut leads back to the first circuits shown in Part (i). We therefore activate and propagate the innermost cut $\widehat{\alpha}\dagger\widehat{u}$ *by applying the rules* $(act\text{-}\mathrm{R})$*,* $(\nwarrow\uparrow I\text{-}ins)$*,* $(\nwarrow\uparrow I\text{-}outs)$*,* $(\nwarrow\uparrow I\text{-}outs)$ *then (Lem. 3.1.21) four times, giving:*

$$\begin{aligned}
&([\widehat{z}\widehat{t}\overline{Q}]\cdot\beta)\widehat{\beta}\dagger\widehat{v}(v\cdot[(([\widehat{x}\widehat{s}\overline{P}]\cdot\alpha)\widehat{\alpha}\dagger\widehat{u}(u\cdot[\overline{R}\widehat{\alpha},\overline{R}\widehat{\alpha}]))\widehat{\beta},(([\widehat{x}\widehat{s}\overline{P}]\cdot\alpha)\widehat{\alpha}\dagger\widehat{u}(u\cdot[\overline{R}\widehat{\alpha},\overline{R}\widehat{\alpha}]))\widehat{\beta}]) \ \to(\uparrow_2)\\
&\quad([\widehat{z}\widehat{t}\overline{Q}]\cdot\beta)\widehat{\beta}\dagger\widehat{v}(v\cdot[(\overline{R}\widehat{\alpha}\dagger\widehat{s}(\overline{R}\widehat{\alpha}\dagger\widehat{x}\overline{P}))\widehat{\beta},(([\widehat{x}\widehat{s}\overline{P}]\cdot\alpha)\widehat{\alpha}\dagger\widehat{u}(u\cdot[\overline{R}\widehat{\alpha},\overline{R}\widehat{\alpha}]))\widehat{\beta}]) \ \to(\uparrow_2)\\
&\quad([\widehat{z}\widehat{t}\overline{Q}]\cdot\beta)\widehat{\beta}\dagger\widehat{v}(v\cdot[(\overline{R}\widehat{\alpha}\dagger\widehat{s}(\overline{R}\widehat{\alpha}\dagger\widehat{x}\overline{P}))\widehat{\beta},(\overline{R}\widehat{\alpha}\dagger\widehat{s}(\overline{R}\widehat{\alpha}\dagger\widehat{x}\overline{P}))\widehat{\beta}]) \ \to(\uparrow_2)\\
&\quad(\overline{R}\widehat{\alpha}\dagger\widehat{s}(\overline{R}\widehat{\alpha}\dagger\widehat{x}\overline{P}))\widehat{\beta}\dagger\widehat{t}((\overline{R}\widehat{\alpha}\dagger\widehat{s}(\overline{R}\widehat{\alpha}\dagger\widehat{x}\overline{P}))\widehat{\beta}\dagger\widehat{z}\overline{Q})
\end{aligned}$$

Again, we eliminate the cuts made with fresh connectors that were introduced by the interpretation. Applying $(act\text{-R})$, $(\curlywedge gc)$, *we get:*

$$\begin{array}{ll} (\overline{R}\widehat{\alpha} \dagger \widehat{s}(\overline{R}\widehat{\alpha} \dagger \widehat{x}\overline{P}))\widehat{\beta} \dagger \widehat{z}\overline{Q} & \to (act\text{-R}), (\curlywedge gc) \\ \dfrac{(\overline{R}\widehat{\alpha} \dagger \widehat{x}\overline{P})\widehat{\beta} \dagger \widehat{z}\overline{Q}}{(R\widehat{\alpha} \dagger \widehat{x}P)\widehat{\beta} \dagger \widehat{z}Q} & = \end{array}$$

By exhausting all possible applications of $(\uparrow_1)$ *and* $(\uparrow_2)$ *during a reduction, we find only the right-hand side of* $(\vee_1)$ *is attainable under our encoding.*

Observe that a choice was made when we derived $\Gamma \vdash \Delta, (A\uparrow A)\uparrow(B\uparrow B)$. For comparison, we give our first encoding, plus the alternative, below.

$$\dfrac{\dfrac{\dfrac{\boxed{} \quad \boxed{}}{A\uparrow A, \Gamma \vdash \Delta, B}(\uparrow L) \qquad \dfrac{\boxed{} \quad \boxed{}}{A\uparrow A, \Gamma \vdash \Delta, B}(\uparrow L)}{A\uparrow A, B\uparrow B, \Gamma \vdash \Delta}(\uparrow L)}{\Gamma \vdash \Delta, (A\uparrow A)\uparrow(B\uparrow B)}(\uparrow R)$$

$$\dfrac{\dfrac{\dfrac{\boxed{} \quad \boxed{}}{B\uparrow B, \Gamma \vdash \Delta, A}(\uparrow L) \qquad \dfrac{\boxed{} \quad \boxed{}}{B\uparrow B, \Gamma \vdash \Delta, A}(\uparrow L)}{A\uparrow A, B\uparrow B, \Gamma \vdash \Delta}(\uparrow L)}{\Gamma \vdash \Delta, (A\uparrow A)\uparrow(B\uparrow B)}(\uparrow R)$$

The derivations differ by the order in which the formulas $(A\uparrow A)$ and $(B\uparrow B)$ are built. In the first derivation, the formula $(A\uparrow A)$ is constructed first, where $(B\uparrow B)$ is constructed first in the second. In the corresponding circuits, this permutation is seen by the order in which the two plugs (whose types would correspond to those formulas) bind over the (duplicated) sub-circuit. Compare the two encodings below:

$$\overline{[R\widehat{\alpha}\widehat{\beta}]\cdot\gamma} = [\widehat{u}\widehat{v}(v\cdot[(u\cdot[\overline{R}\widehat{\alpha}, \overline{R}\widehat{\alpha}])\widehat{\beta}, (u\cdot[\overline{R}\widehat{\alpha}, \overline{R}\widehat{\alpha}])\widehat{\beta}])]\cdot\gamma$$

or

$$\overline{[R\widehat{\alpha}\widehat{\beta}]\cdot\gamma} = [\widehat{u}\widehat{v}(v\cdot[(u\cdot[\overline{R}\widehat{\beta}, \overline{R}\widehat{\beta}])\widehat{\alpha}, (u\cdot[\overline{R}\widehat{\beta}, \overline{R}\widehat{\beta}])\widehat{\alpha}])]\cdot\gamma$$

We find that we can simulate the rule $(\vee_2)$ if we use the alternative interpretation of the 'or output' circuit. This result is shown below.

Lemma 5.4.13 (Simulation of $\mathcal{X}^{\vee}$ in $\mathcal{X}^{\uparrow}$ using alternative encoding) *Using an alternative encoding*

$$\begin{aligned} & \overline{([R\widehat{\alpha}\widehat{\beta}]\cdot\gamma)\widehat{\gamma} \dagger \widehat{y}(y\cdot[\widehat{x}P, \widehat{z}Q])} \\ = \ & \overline{([R\widehat{\alpha}\widehat{\beta}]\cdot\gamma)}\widehat{\gamma} \dagger \widehat{y}\overline{(y\cdot[\widehat{x}P, \widehat{z}Q])} \\ = \ & ([\widehat{u}\widehat{v}(u\cdot[(v\cdot[\overline{R}\widehat{\beta}, \overline{R}\widehat{\beta}])\widehat{\alpha}, (v\cdot[\overline{R}\widehat{\beta}, \overline{R}\widehat{\beta}])\widehat{\alpha}])]\cdot\gamma)\widehat{\gamma} \dagger \widehat{y}(y\cdot[([\widehat{x}\widehat{s}\overline{P}]\cdot\mu)\widehat{\mu}, ([\widehat{z}\widehat{t}\overline{Q}]\cdot\delta)\widehat{\delta}]) \end{aligned} \quad (5.6)$$

Where u, v, s, t, δ, μ are fresh.

$$
\begin{array}{l}
([\widehat{u}\widehat{v}(u\cdot[(v\cdot[\overline{R}\widehat{\beta},\overline{R}\widehat{\beta}])\widehat{\alpha},(v\cdot[\overline{R}\widehat{\beta},\overline{R}\widehat{\beta}])\widehat{\alpha}])]\cdot\gamma)\widehat{\gamma}\dagger\widehat{y}(y\cdot[([\widehat{x}\widehat{s}\overline{P}]\cdot\mu)\widehat{\mu},([\widehat{z}\widehat{t}\overline{Q}]\cdot\delta)\widehat{\delta}]) \\
([\widehat{x}\widehat{s}\overline{P}]\cdot\mu)\widehat{\mu}\dagger\widehat{u}(([\widehat{z}\widehat{t}\overline{Q}]\cdot\delta)\widehat{\delta}\dagger\widehat{v}(u\cdot[(v\cdot[\overline{R}\widehat{\beta},\overline{R}\widehat{\beta}])\widehat{\alpha},(v\cdot[\overline{R}\widehat{\beta},\overline{R}\widehat{\beta}])\widehat{\alpha}])) \\
([\widehat{x}\widehat{s}\overline{P}]\cdot\mu)\widehat{\mu}\dagger\widehat{u}(u\cdot[(([\widehat{z}\widehat{t}\overline{Q}]\cdot\delta)\widehat{\delta}\dagger\widehat{v}(v\cdot[\overline{R}\widehat{\beta},\overline{R}\widehat{\beta}]))\widehat{\alpha},(([\widehat{z}\widehat{t}\overline{Q}]\cdot\delta)\widehat{\delta}\dagger\widehat{v}(v\cdot[\overline{R}\widehat{\beta},\overline{R}\widehat{\beta}]))\widehat{\alpha}]) \\
([\widehat{x}\widehat{s}\overline{P}]\cdot\mu)\widehat{\mu}\dagger\widehat{u}(u\cdot[(\overline{R}\widehat{\beta}\dagger\widehat{z}(\overline{R}\widehat{\beta}\dagger\widehat{t}\overline{Q}))\widehat{\alpha},(([\widehat{z}\widehat{t}\overline{Q}]\cdot\delta)\widehat{\delta}\dagger\widehat{v}(v\cdot[\overline{R}\widehat{\beta},\overline{R}\widehat{\beta}]))\widehat{\alpha}]) \\
([\widehat{x}\widehat{s}\overline{P}]\cdot\mu)\widehat{\mu}\dagger\widehat{u}(u\cdot[(\overline{R}\widehat{\beta}\dagger\widehat{z}(\overline{R}\widehat{\beta}\dagger\widehat{t}\overline{Q}))\widehat{\alpha},(\overline{R}\widehat{\beta}\dagger\widehat{z}(\overline{R}\widehat{\beta}\dagger\widehat{t}\overline{Q}))\widehat{\alpha}]) \\
(\overline{R}\widehat{\beta}\dagger\widehat{z}(\overline{R}\widehat{\beta}\dagger\widehat{t}\overline{Q}))\widehat{\alpha}\dagger\widehat{x}((\overline{R}\widehat{\beta}\dagger\widehat{z}(\overline{R}\widehat{\beta}\dagger\widehat{t}\overline{Q}))\widehat{\alpha}\dagger\widehat{s}\overline{P}) \\
(\overline{R}\widehat{\beta}\dagger\widehat{z}\overline{Q})\widehat{\alpha}\dagger\widehat{x}((\overline{R}\widehat{\beta}\dagger\widehat{z}(\overline{R}\widehat{\beta}\dagger\widehat{t}\overline{Q}))\widehat{\alpha}\dagger\widehat{s}\overline{P}) \\
(\overline{R}\widehat{\beta}\dagger\widehat{z}\overline{Q})\widehat{\alpha}\dagger\widehat{x}((\overline{R}\widehat{\beta}\dagger\widehat{z}\overline{Q})\widehat{\alpha}\dagger\widehat{s}\overline{P}) \\
\hline
(\overline{R}\widehat{\beta}\dagger\widehat{z}\overline{Q})\widehat{\alpha}\dagger\widehat{x}\overline{P} \\
\hline
(R\widehat{\beta}\dagger\widehat{z}Q)\widehat{\alpha}\dagger\widehat{x}P
\end{array}
$$

This is the interpretation of the right-hand side of the rule $(\vee_2)$.

The main result of this section is that logical expressivity does not necessarily imply computational expressivity. That is, the ability of a set of connectives to logically express another set of connectives does not imply that a term calculus built from the former set will be able to simulate all of the reductions in the term calculus built from the latter set. This has been shown by counter examples.

5.5 Interpreting 'if-and-only-if'

Part of the work presented in this section was completed in collaboration with Alexander J. Summers [74].

In this section we study the computational behaviour of the logical connective 'if-and-only-if' ('iff' for short) that evaluates to *true* exactly when its two arguments have the same truth value. We could equally have chosen to study the negation of this connective 'exclusive-or', whose $\mathcal{X}$-style term representations will be almost the same except that the free connector that is introduced in each term will be of the opposite kind (input versus output).

We will follow the steps of the recipe outlined in Section 5.2 and build a Curry-Howard pair of calculi based on the 'iff' connective. We first extract the introduction rules for the connective (written $A{\leftrightarrow}B$), by building partial derivations of a logically equivalent formula, say $\neg(A\vee B)\vee(A\wedge B)$. The pair of rules obtained are:

$$\frac{\Gamma\vdash\Delta,A,B \qquad A,B,\Gamma\vdash\Delta}{A{\leftrightarrow}B,\Gamma\vdash\Delta}\,(\leftrightarrow L) \qquad \frac{A,\Gamma\vdash\Delta,B \qquad B,\Gamma\vdash\Delta,A}{\Gamma\vdash\Delta,A{\leftrightarrow}B}\,(\leftrightarrow R)$$

Comparing these rules to those discussed in the previous sections, we observe that the rules each bind *two* inputs and *two* outputs, and each rule has two sub-proofs. This yields the following (relatively complex) input and output circuits: $y{\cdot}[M\widehat{\mu}\widehat{\sigma}, \widehat{i}\widehat{j}N]$ and $[\widehat{x}P\widehat{\alpha}, \widehat{z}Q\widehat{\delta}]{\cdot}\gamma$.

The propagation and renaming rules for a Curry-Howard pair of calculi built on this connective are straightforward to define. The challenge is in defining the principal reduction rule.

The principal reduction rule for 'iff' should transform a proof that cuts together an $(\leftrightarrow R)$ formula with an $(\leftrightarrow L)$ formula, or using the terminology of circuits, the rule should eliminate the 'iff' circuit constructors from the following, (where γ, y are introduced),

$$([\widehat{x}P\widehat{\alpha}, \widehat{z}Q\widehat{\delta}]{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{y}(y{\cdot}[M\widehat{\mu}\widehat{\sigma}, \widehat{i}\widehat{j}N])$$

The right-hand side of the rule is not straightforward to determine. In the following discussion, we will reason about what a suitable right-hand side of the principal reduction rule for 'iff' might look like.

First we remark on the striking resemblance between these terms and the $\mathcal{X}$-calculus syntax used to represent the implication connective. The output circuit is reminiscent of the export, except two 'functions' are available over the same output rather than one (n.b., $A{\leftrightarrow}B{\equiv}(A{\rightarrow}B){\wedge}(B{\rightarrow}A)$). The input circuit is reminiscent of an import with two binders over each of its proper sub-circuits instead of one.

In the case of an import, say $R\widehat{\psi}\,[l]\,\widehat{k}S$, a connection between the sub-circuits R and S is sought via the bound connectors. In general, connecting ψ to k directly would result in the restriction that 'implications' must typed with $A{\rightarrow}A$. It is the body of an export which must be inserted in between the subterms of the 'iff' input circuit that allows the more general type of $A{\rightarrow}B$. If we think of the input circuit for 'iff' as a kind of import, the problem we must solve is again that of connecting outputs and inputs between the terms M and N.

Recall that formulas in the antecedent part of a sequent are read conjunctively, while formulas in the succedent part are read disjunctively. This interpretation carries over to the circuits. In the input circuit $y{\cdot}[M\widehat{\mu}\widehat{\sigma}, \widehat{i}\widehat{j}N]$, M offers a value of type A *or* a value of type B (loosely a value of type $A{\vee}B$), while N requires both a value of type A *and* a value of type B (loosely, requires a value of type $A{\wedge}B$). Therefore, the problem we must solve in trying to join these two proofs is essentially that of determining how we can convert from a value of type $A{\vee}B$ to a value of type $A{\wedge}B$. That is, we intuitively need to construct a circuit of type

$(A \vee B) \rightarrow (A \wedge B)$. Note that this 'intuitive' formula is actually logically equivalent to $A \leftrightarrow B$, which is the kind of functionality provided on γ by the 'iff' output circuit.

We return to the previous method of determining the principal reduction rule as detailed in Section 5.2, i.e., that of considering how one would reduce a cut between derivations that introduce a formula logically equivalent to $A \leftrightarrow B$. We cut together the proofs that derive $\neg(A \vee B) \vee (A \wedge B)$ on the left and right of the sequent, then reduce them using the cut-elimination rules for negation, disjunction and conjunction. The process allows us to extract the following right-hand side, (with k, w fresh),

$$((M\widehat{\mu} \dagger \widehat{x}P)\widehat{\sigma} \dagger \widehat{k}\langle k \cdot \alpha \rangle)\widehat{\alpha} \dagger \widehat{j}(((M\widehat{\sigma} \dagger \widehat{z}Q)\widehat{\mu} \dagger \widehat{w}\langle w \cdot \delta \rangle)\widehat{\delta} \dagger \widehat{i}N)$$

This is better understood in a diagrammatic form where the types of the reduct can be seen (see Figure 5.5). The twisting of wires represents an (implicit) contraction in the proof, which 'merges' two connections (occurrences of the same formula) into one. The circuit P is used to convert the type of one of the outputs of M, so that both of M's outputs end up with the same type. The cut with a capsule is used to rename the other output of M to α (the same name as the output of P) so that they can be contracted into one. In this way, we can connect the two outputs of M to a single input of N via a cut. Making a copy of the term M allows us to simultaneously connect to both inputs of N. Without the two copies, it is difficult to construct cuts that make all of these connections.

An alternative and symmetrical right-hand side that could be built using the same process is shown below, (with π, τ fresh),

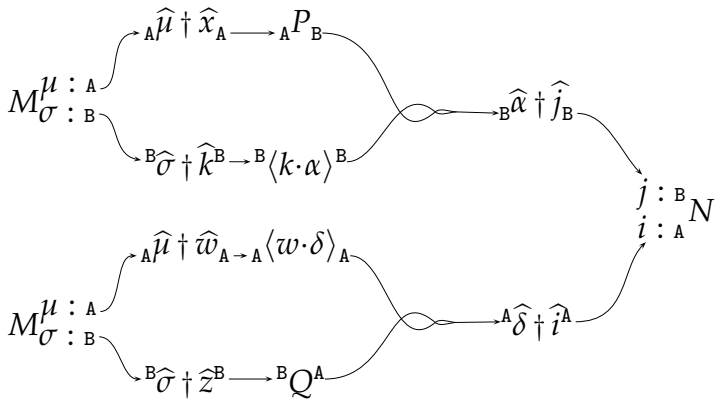
$$(M\widehat{\mu} \dagger \widehat{x}(\langle x \cdot \pi \rangle \widehat{\pi} \dagger \widehat{i}(P\widehat{\alpha} \dagger \widehat{j}N)))\widehat{\sigma} \dagger \widehat{z}(\langle z \cdot \tau \rangle \widehat{\tau} \dagger \widehat{j}(Q\widehat{\delta} \dagger \widehat{i}N))$$

With this alternative, two copies of N (rather than M) are made and inputs are renamed rather than outputs. We are able to condense the connection diagram of Figure 5.5 into a form which focuses on the direct connections made via each cut (see Figure 5.6). We give a more formal definition of the principal reduction rules below.

Definition 5.5.1 (Principal iff-reduction rule with copying) *The term*

$$([\widehat{x}P\widehat{\alpha}, \widehat{z}Q\widehat{\delta}] \cdot \gamma)\widehat{\gamma} \dagger \widehat{y}(y \cdot [M\widehat{\mu}\widehat{\sigma}, \widehat{i}\widehat{j}N])$$

where, γ, y are introduced, reduces to one of the following variants (with k, w, π, τ



$$((M\widehat{\mu} \dagger \widehat{x})\widehat{\sigma} \dagger \widehat{k}\langle k{\cdot}\alpha\rangle)\widehat{\alpha} \dagger \widehat{j}(((M\widehat{\sigma} \dagger \widehat{z}Q)\widehat{\mu} \dagger \widehat{w}\langle w{\cdot}\delta\rangle)\widehat{\delta} \dagger \widehat{i}N)$$

Figure 5.5: A Possible Right-Hand Side for the 'iff' Principal Reduction Rule.

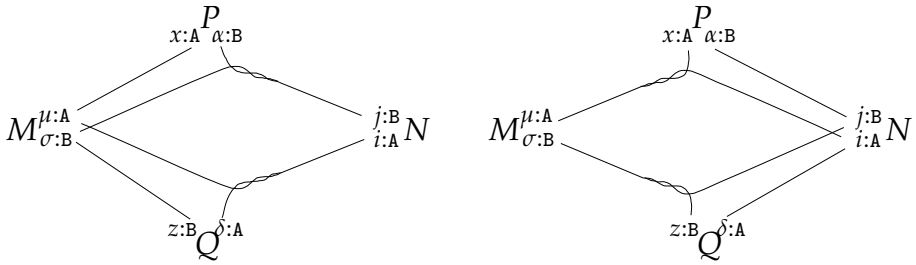


Figure 5.6: Simplified Connection Diagrams for Definition 5.5.1

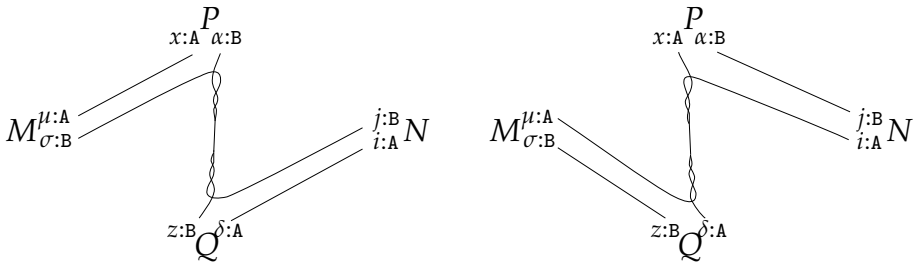


Figure 5.7: Simplified Connection Diagrams for the Reducts of Definition 5.5.2

fresh).

$$(\leftrightarrow_{c1}): \quad ((M\widehat{\mu} \dagger \widehat{x}P)\widehat{\sigma} \dagger \widehat{k}\langle k{\cdot}\alpha\rangle)\widehat{\alpha} \dagger \widehat{j}(((M\widehat{\sigma} \dagger \widehat{z}Q)\widehat{\mu} \dagger \widehat{w}\langle w{\cdot}\delta\rangle)\widehat{\delta} \dagger \widehat{i}N)$$
$$(\leftrightarrow_{c2}): \quad (M\widehat{\mu} \dagger \widehat{x}(\langle x{\cdot}\pi\rangle\widehat{\pi} \dagger \widehat{i}(P\widehat{\alpha} \dagger \widehat{j}N)))\widehat{\sigma} \dagger \widehat{z}(\langle z{\cdot}\tau\rangle\widehat{\tau} \dagger \widehat{j}(Q\widehat{\delta} \dagger \widehat{i}N))$$

As mentioned previously, a copy of either M or N is used to facilitate the connection of each output of M to each input of N. The question arises of whether this copying is necessary. One of the graphs of Figure 5.6 renames both outputs of M while the other renames both inputs of N.

In exploring other ways in which M and N could be connected we were particularly interested in determining whether it would be possible to obtain a right-hand side which did not require copying. We sought to distribute the connections in a more symmetrical fashion because we believed that the copying was only necessary due to the large number of connections being made with one term or the other. We discovered a solution where we rename *one* output in M and *one* input in N. This leads to the connection diagrams shown in Figure 5.7. The reader can verify that a path exists from each output of M to each input of N.

This leads us to a simpler definition for the principal logical rule.

Definition 5.5.2 (Simplified Principal iff-reduction Rule) *The term*

$$([\widehat{x}P\widehat{\alpha}, \widehat{z}Q\widehat{\delta}]{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{y}(y{\cdot}[M\widehat{\mu}\widehat{\sigma}, \widehat{i}\widehat{j}N])$$

where, γ, y introduced and k, π fresh, reduces to one of the following variants.

$$(\leftrightarrow_1): \quad ((M\widehat{\mu} \dagger \widehat{x}P)\widehat{\sigma} \dagger \widehat{k}\langle k{\cdot}\alpha\rangle)\widehat{\alpha} \dagger \widehat{z}(\langle z{\cdot}\pi\rangle\widehat{\pi} \dagger \widehat{j}(Q\widehat{\delta} \dagger \widehat{i}N))$$
$$(\leftrightarrow_2): \quad ((M\widehat{\sigma} \dagger \widehat{z}Q)\widehat{\mu} \dagger \widehat{k}\langle k{\cdot}\delta\rangle)\widehat{\delta} \dagger \widehat{x}(\langle x{\cdot}\pi\rangle\widehat{\pi} \dagger \widehat{i}(P\widehat{\alpha} \dagger \widehat{j}N))$$

These reducts will be significantly cheaper to evaluate than those given in Definition 5.5.1 since an extra copy of M (or N) is not required and fewer cuts are needed to represent all the necessary connections. In the following, we will use the simplified principal reduction rules for 'iff'. We give the full definition of the $\mathcal{X}^{\leftrightarrow}$ calculus in Figure 5.8.

Definition 5.5.3 ($\mathcal{X}^{\leftrightarrow}$-Syntax) *The circuits of the $\mathcal{X}^{\leftrightarrow}$-calculus are defined by the following grammar, where x, y, z, i, j range over the infinite set of* sockets*, and $\alpha, \beta, \delta, \gamma, \sigma$ over* plugs*.*

$$\begin{array}{rcccccc} M, N ::= & \langle x{\cdot}\alpha\rangle & \mid & z{\cdot}[M\widehat{\mu}\widehat{\sigma}, \widehat{i}\widehat{j}N] & \mid & [\widehat{x}M\widehat{\alpha}, \widehat{z}N\widehat{\delta}]{\cdot}\gamma & \mid M\widehat{\alpha} \dagger \widehat{x}N \\ & \text{axiom} & & \text{`iff' input circuit} & & \text{`iff' output circuit} & \text{cut} \end{array}$$

Definition 5.5.4 (Typing Rules for $\mathcal{X}^{\leftrightarrow}$) *The axiom and cut are typed as usual (Definition 5.1.3). The input and output circuits for 'iff' are typed as follows.*

$$\frac{M :\cdot \Gamma \vdash \mu{:}A, \alpha{:}B, \Delta \qquad N :\cdot \Gamma, i{:}A, j{:}B \vdash \Delta}{z{\cdot}[M\widehat{\mu}\widehat{\sigma}, \widehat{i}\widehat{j}N] :\cdot \Gamma, z{:}(A{\leftrightarrow}B) \vdash \Delta}\,(\leftrightarrow L)$$

$$\frac{M :\cdot \Gamma, x{:}A \vdash \alpha{:}B, \Delta \qquad N :\cdot \Gamma, z{:}B \vdash \delta{:}A, \Delta}{[\widehat{x}M\widehat{\alpha}, \widehat{z}N\widehat{\delta}]{\cdot}\gamma :\cdot \Gamma \vdash \gamma{:}(A{\leftrightarrow}B), \Delta}\,(\leftrightarrow R)$$

Definition 5.5.5 ($\mathcal{X}^{\leftrightarrow}$ Reduction Rules) *We extend Definition 5.2.4 with,*

***Left Propagation Rules** :*

$$\begin{array}{rl} (\leftrightarrow\text{O-outs}\not\prec): & ([\widehat{x}M\widehat{\alpha}, \widehat{z}N\widehat{\delta}]{\cdot}\gamma)\widehat{\gamma} \not\prec \widehat{y}R \;\to\; ([\widehat{x}(M\widehat{\gamma} \not\prec \widehat{y}R)\widehat{\alpha}, \widehat{z}(N\widehat{\gamma} \not\prec \widehat{y}R)\widehat{\delta}]{\cdot}\mu)\widehat{\mu} \dagger \widehat{y}R \\ (\leftrightarrow\text{O-ins}\not\prec): & ([\widehat{x}M\widehat{\alpha}, \widehat{z}N\widehat{\delta}]{\cdot}\pi)\widehat{\gamma} \not\prec \widehat{y}R \;\to\; [\widehat{x}(M\widehat{\gamma} \not\prec \widehat{y}R)\widehat{\alpha}, \widehat{z}(N\widehat{\gamma} \not\prec \widehat{y}R)\widehat{\delta}]{\cdot}\pi \leftarrow \gamma{\neq}\pi \\ (\leftrightarrow I\not\prec): & (z{\cdot}[M\widehat{\mu}\widehat{\sigma}, \widehat{i}\widehat{j}N])\widehat{\gamma} \not\prec \widehat{y}R \;\to\; z{\cdot}[(M\widehat{\gamma} \not\prec \widehat{y}R)\widehat{\mu}\widehat{\sigma}, \widehat{i}\widehat{j}(N\widehat{\gamma} \not\prec \widehat{y}R)] \end{array}$$

***Right Propagation Rules** :*

$$\begin{array}{rl} (\not\leftarrow\!\leftrightarrow\text{I-outs}): & R\widehat{\gamma} \not\leftarrow \widehat{y}(y{\cdot}[M\widehat{\mu}\widehat{\sigma}, \widehat{i}\widehat{j}N]) \;\to\; R\widehat{\gamma} \dagger \widehat{y}(y{\cdot}[(R\widehat{\gamma} \not\leftarrow \widehat{y}M)\widehat{\mu}\widehat{\sigma}, \widehat{i}\widehat{j}(R\widehat{\gamma} \not\leftarrow \widehat{y}N)]) \\ (\not\leftarrow\!\leftrightarrow\text{I-ins}): & R\widehat{\gamma} \not\leftarrow \widehat{y}(z{\cdot}[M\widehat{\mu}\widehat{\sigma}, \widehat{i}\widehat{j}N]) \;\to\; z{\cdot}[(R\widehat{\gamma} \not\leftarrow \widehat{y}M)\widehat{\mu}\widehat{\sigma}, \widehat{i}\widehat{j}(R\widehat{\gamma} \not\leftarrow \widehat{y}N)] \leftarrow y{\neq}z \\ (\not\leftarrow\!\leftrightarrow O): & R\widehat{\gamma} \not\leftarrow \widehat{y}([\widehat{x}P\widehat{\alpha}, \widehat{z}Q\widehat{\delta}]{\cdot}\mu) \;\to\; [\widehat{x}(R\widehat{\gamma} \not\leftarrow \widehat{y}P)\widehat{\alpha}, \widehat{z}(R\widehat{\gamma} \not\leftarrow \widehat{y}Q)\widehat{\delta}]{\cdot}\mu \end{array}$$

***Renaming Rules** :*

$$\begin{array}{rl} (\leftrightarrow\text{I-rn}): & ([\widehat{x}P\widehat{\alpha}, \widehat{z}Q\widehat{\delta}]{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{y}\langle y{\cdot}\mu\rangle \;\to\; [\widehat{x}P\widehat{\alpha}, \widehat{z}Q\widehat{\delta}]{\cdot}\mu \leftarrow \gamma, y \textit{ introduced} \\ (\leftrightarrow\text{O-rn}): & \langle z{\cdot}\gamma\rangle\widehat{\gamma} \dagger \widehat{y}(y{\cdot}[M\widehat{\mu}\widehat{\sigma}, \widehat{i}\widehat{j}N]) \;\to\; z{\cdot}[M\widehat{\mu}\widehat{\sigma}, \widehat{i}\widehat{j}N] \leftarrow \gamma, y \textit{ introduced} \end{array}$$

***Principal Reduction Rules** :*

$$\begin{array}{rl} (\leftrightarrow_1): & ((M\widehat{\mu} \dagger \widehat{x}P)\widehat{\sigma} \dagger \widehat{k}\langle k{\cdot}\alpha\rangle)\widehat{\alpha} \dagger \widehat{j}(((M\widehat{\sigma} \dagger \widehat{z}Q)\widehat{\mu} \dagger \widehat{w}\langle w{\cdot}\delta\rangle)\widehat{\delta} \dagger \widehat{i}N) \leftarrow \gamma, y \textit{ introduced} \\ (\leftrightarrow_2): & (M\widehat{\mu} \dagger \widehat{x}(\langle x{\cdot}\pi\rangle\widehat{\pi} \dagger \widehat{i}(P\widehat{\alpha} \dagger \widehat{j}N)))\widehat{\sigma} \dagger \widehat{z}(\langle z{\cdot}\tau\rangle\widehat{\tau} \dagger \widehat{j}(Q\widehat{\delta} \dagger \widehat{i}N)) \leftarrow \gamma, y \textit{ introduced} \end{array}$$

Figure 5.8: The $\mathcal{X}^{\leftrightarrow}$-Calculus

5.5.1 Simulating other connectives with 'iff'

In Section 5.4.1, we demonstrated that the logical expressivity of a connective does not imply its computational expressivity (i.e., a calculus may not be able to simulate the reductions of calculi built from connectives which it can logically express). The only logical connectives expressible by 'iff' are $\top$ and *id*. Since 'iff' does not have a great amount of logical expressivity, this might (with the result of the previous section) lead us to believe its simulation capabilities are limited. However, we find this is not the case; in fact we are able to simulate the reductions associated with several other connectives, i.e., we can encode the syntax for these other connectives in such a way that reductions are preserved.

If we look at the 'iff' circuits themselves, we find they provide a wealth of input and output connectors arranged in different combinations over a number of subterms. We also observe that the principal reduction rules offer a number of interactions between these different subterms, giving scope for modelling a variety of computational behaviour, some of which may be new.

As an example of a connective which can be computationally expressed (but not logically expressed) by 'iff', we show how to express the syntax and reduction behaviour of the $\mathcal{X}$-calculus (based on the implication connective) in the $\mathcal{X}^{\leftrightarrow}$-calculus.

As remarked earlier, the 'iff' input circuit is reminiscent of an import with two binders over each of its subterms rather than one, and the 'iff' output circuit is reminiscent of an export, except that two 'functions' are available over the same interface rather than one. With this observation in mind, we move towards an encoding of the $\mathcal{X}$-calculus into $\mathcal{X}^{\leftrightarrow}$.

We can sensibly assume that when encoding the export $\widehat{x}P\widehat{\alpha}\cdot\gamma$ into the 'iff' output circuit $[\widehat{x}P\widehat{\alpha},\widehat{z}Q\widehat{\delta}]\cdot\gamma$, we require only *one* of the two subterms, say P. This leaves the question of what we should do with Q. By making Q the capsule $\langle y\cdot\delta\rangle$, with $y\neq z$, we can give an encoding that is sound (no undesired reductions are possible) providing that we restrict the reduction to always use the principal logical rule $(\leftrightarrow_1)$ given in Figure 5.8. One might view this as a strategy on the reduction (one always has the choice of which variant of the principal 'iff' rule to use). Our encoding is as follows.

Definition 5.5.6 (Interpretation of $\mathcal{X}$ into $\mathcal{X}^{\leftrightarrow}$)

$$\begin{array}{rcll}
\overleftrightarrow{\langle x{\cdot}\alpha\rangle} & = & \langle x{\cdot}\alpha\rangle & \\
\overleftrightarrow{\widehat{x}P\widehat{\alpha}{\cdot}\gamma} & = & [\widehat{x}\,\overleftrightarrow{P}\,\widehat{\alpha}, \widehat{z}\langle y{\cdot}\delta\rangle\widehat{\delta}]{\cdot}\gamma & z, y, \delta \textit{ fresh} \\
\overleftrightarrow{M\widehat{\alpha}\,[y]\,\widehat{x}N} & = & y{\cdot}[\,\overleftrightarrow{M}\,\widehat{\alpha}\widehat{\beta}, \widehat{z}\widehat{x}\,\overleftrightarrow{N}\,] & \beta, z \textit{ fresh} \\
\overleftrightarrow{M\widehat{\alpha}\dagger\widehat{x}N} & = & \overleftrightarrow{M}\,\widehat{\alpha}\dagger\widehat{x}\,\overleftrightarrow{N} &
\end{array}$$

Notice that in the interpretation of $\widehat{x}P\widehat{\alpha}{\cdot}\gamma$, had we chosen Q (the right-hand subterm) to be $\langle z{\cdot}\delta\rangle$, this would have forced the types for z and δ, and therefore x and α to be the same. As a result, our encoding would not preserve typeability, since in the original term x and α need not have had the same type.

In fact, the type derivations in the two systems are closely related; one can define a further encoding from a type-derivation for P in the $\mathcal{X}$-calculus to a type-derivation for $\overleftrightarrow{P}$ in the corresponding $\mathcal{X}^{\leftrightarrow}$ system. This encoding is given below.

Definition 5.5.7 (Encoding of Contexts)

$$\begin{array}{rcl}
\overleftrightarrow{\phi} & = & \phi \\
\overleftrightarrow{A_1 \rightarrow A_2} & \rightarrow & \overleftrightarrow{A_1} \rightarrow \overleftrightarrow{A_2} \\
\overleftrightarrow{A_1 \leftrightarrow A_2} & \rightarrow & \overleftrightarrow{A_1} \rightarrow \overleftrightarrow{A_2}
\end{array}$$

We extend the above encoding to work on contexts as follows,

$$\begin{array}{rcl}
\overleftrightarrow{\Gamma} & \rightarrow & \left\{ x{:}\,\overleftrightarrow{A} \;\mid\; x{:}A \in \Gamma \right\} \\
\overleftrightarrow{\Delta} & \rightarrow & \left\{ \alpha{:}\,\overleftrightarrow{A} \;\mid\; \alpha{:}A \in \Delta \right\}
\end{array}$$

We have the following result for our encoding.

Theorem 5.5.8 (Preservation of typeability) *For any $\mathcal{X}$-term P, P is typeable 'iff' $\overleftrightarrow{P}$ is typeable.*

Proof 5.5.9 *In the following, we will use the symbols $\vdash$ and $\vdash_{\leftrightarrow}$ to distinguish between the type systems of the $\mathcal{X}$-calculus and the $\mathcal{X}^{\leftrightarrow}$-calculus respectively.*

The proof is in two parts: (a) left-to right and (b) right-to-left.

(a). First we show, by the induction on the structure of $\mathcal{X}$-circuits P, that if $P :\cdot \Gamma \vdash \Delta$, then for some Γ', $\overleftrightarrow{P} :\cdot \overleftrightarrow{\Gamma'} \vdash_{\leftrightarrow} \overleftrightarrow{\Delta}$

$P \equiv \langle x{\cdot}\alpha \rangle$: *This can only be typed in* $\vdash$ *using the rule* (Ax)*, i.e., for some types* A, B *and contexts* Γ, Δ*:*

$$\frac{}{\langle x{\cdot}\alpha \rangle \;:\cdot\; x{:}A, \Gamma \vdash \Delta, \alpha{:}A}\,(Ax)$$

Our encoding gives us $\overleftrightarrow{P} = \langle x{\cdot}\alpha \rangle$*, which can only be typed in* $\vdash_{\leftrightarrow}$ *with*

$$\frac{}{\langle x{\cdot}\alpha \rangle \;:\cdot\; x{:}\overleftrightarrow{A}, \overleftrightarrow{\Gamma} \;\vdash_{\leftrightarrow}\; \overleftrightarrow{\Delta}, \alpha{:}\overleftrightarrow{A}}\,(Ax)$$

$P \equiv \widehat{x}Q\widehat{\alpha}{\cdot}\beta$: *The export can only be typed in* $\vdash$ *using* $(\rightarrow R)$*, i.e., for some types* A, B *and contexts* Γ, Δ*,*

$$\frac{Q \;:\cdot\; x{:}A, \Gamma \vdash \Delta, \alpha{:}B}{\widehat{x}Q\widehat{\alpha}{\cdot}\beta \;:\cdot\; \Gamma \vdash \Delta, \beta{:}A{\rightarrow}B}\,(\rightarrow R)$$

By induction, for some Γ'*,* $\overleftrightarrow{Q} \;:\cdot\; x{:}\overleftrightarrow{A}, \overleftrightarrow{\Gamma} \cup \Gamma' \;\vdash_{\leftrightarrow}\; \overleftrightarrow{\Delta}, \alpha{:}\overleftrightarrow{B}$*. By weakening as appropriate and letting* $\Gamma'' = \overleftrightarrow{\Gamma} \cup \Gamma'$*, we can construct the following typing derivation for* $\overleftrightarrow{P}$*.*

$$\frac{\overleftrightarrow{Q} \;:\cdot\; x{:}\overleftrightarrow{A}, \Gamma'' \;\vdash_{\leftrightarrow}\; \overleftrightarrow{\Delta}, \alpha{:}\overleftrightarrow{B} \qquad \dfrac{}{\langle y{\cdot}\delta \rangle \;:\cdot\; y{:}\overleftrightarrow{A}, z{:}\overleftrightarrow{B}, \Gamma'' \;\vdash_{\leftrightarrow}\; \overleftrightarrow{\Delta}, \delta{:}\overleftrightarrow{A}}\,(Ax)}{[\widehat{x}\overleftrightarrow{Q}\widehat{\alpha}, \widehat{z}\langle y{\cdot}\delta \rangle \widehat{\delta}]{\cdot}\gamma \;:\cdot\; y{:}\overleftrightarrow{A}, \Gamma'' \;\vdash_{\leftrightarrow}\; \overleftrightarrow{\Delta}, \gamma{:}\overleftrightarrow{A} \leftrightarrow \overleftrightarrow{B}}\,(\leftrightarrow R)$$

i.e., there exists some Γ''' *such that* $P \;:\cdot\; \Gamma''' \cup \overleftrightarrow{\Gamma} \;\vdash_{\leftrightarrow}\; \overleftrightarrow{\Delta}, \gamma{:}\overleftrightarrow{A} \leftrightarrow \overleftrightarrow{B}$ *and* $\Gamma''' = \Gamma' \cup y{:}\overleftrightarrow{A}$*.*

$P \equiv M\widehat{\alpha}\,[z]\,\widehat{x}N$: *The import can only be typed using* $(\rightarrow L)$*, i.e.,*

$$\frac{M \;:\cdot\; \Gamma \vdash \Delta, \alpha{:}A \qquad N \;:\cdot\; x{:}B, \Gamma \vdash \Delta}{M\widehat{\alpha}\,[z]\,\widehat{x}N \;:\cdot\; z{:}A{\rightarrow}B, \Gamma \vdash \Delta}\,(\rightarrow L)$$

By induction twice, and for some Γ', Γ''*, we have:*

$$\begin{array}{l} M \;:\cdot\; \Gamma' \cup \overleftrightarrow{\Gamma} \;\vdash_{\leftrightarrow}\; \overleftrightarrow{\Delta}, \alpha{:}\overleftrightarrow{A} \\ N \;:\cdot\; x{:}\overleftrightarrow{A}, \Gamma'' \cup \overleftrightarrow{\Gamma} \;\vdash_{\leftrightarrow}\; \overleftrightarrow{\Delta} \end{array}$$

$P \equiv M\widehat{\alpha} \dagger \widehat{x}N$: *The import can only be typed using* (Cut)*, i.e.,*

$$\frac{M \;:\cdot\; \Gamma \;\vdash_{\leftrightarrow}\; \Delta, \alpha{:}A \qquad N \;:\cdot\; x{:}B, \Gamma \;\vdash_{\leftrightarrow}\; \Delta}{M\widehat{\alpha} \dagger \widehat{x}N \;:\cdot\; \Gamma \;\vdash_{\leftrightarrow}\; \Delta}\,(\rightarrow L)$$

By induction twice, and for some Γ', Γ'', we have:

$$M :\cdot \Gamma' \cup \overleftrightarrow{\Gamma} \vdash_{\leftrightarrow} \overleftrightarrow{\Delta}, \alpha{:}\overleftrightarrow{A}$$
$$N :\cdot x{:}\overleftrightarrow{A}, \Gamma'' \cup \overleftrightarrow{\Gamma} \vdash_{\leftrightarrow} \overleftrightarrow{\Delta}$$

And letting $\Gamma''' = \Gamma' \cup \Gamma'' \cup \overleftrightarrow{\Gamma}$, we can construct a typing derivation for $\overleftrightarrow{P}$:

$$\frac{M :\cdot \alpha{:}\overleftrightarrow{A}, \Gamma''' \vdash_{\leftrightarrow} \overleftrightarrow{\Delta} \quad N :\cdot x{:}\overleftrightarrow{A}, \Gamma''' \vdash_{\leftrightarrow} \overleftrightarrow{\Delta}}{\overleftrightarrow{M}\,\widehat{\alpha}\,\dagger\,\widehat{x}\,\overleftrightarrow{N} :\cdot \Gamma''' \vdash_{\leftrightarrow} \overleftrightarrow{\Delta}}\,(Cut)$$

(b). Now we show, by induction on the structure of $\mathcal{X}$-circuits P, that if $\overleftrightarrow{P} :\cdot \Gamma \vdash_{\leftrightarrow} \Delta$, then $P :\cdot \overleftrightarrow{\Gamma} \vdash \overleftrightarrow{\Delta}$.

$P \equiv \langle x{\cdot}\alpha \rangle$ *: By our encoding, we have $\overleftrightarrow{P} = \langle x{\cdot}\alpha \rangle$. The capsule can only be typed with the rule (Ax), i.e.,*

$$\frac{}{\langle x{\cdot}\alpha \rangle :\cdot x{:}A, \Gamma \vdash_{\leftrightarrow} \Delta, \alpha{:}A}\,(Ax)$$

By encoding contexts, we can type P with:

$$\frac{}{\langle x{\cdot}\alpha \rangle :\cdot x{:}\overleftrightarrow{A}, \overleftrightarrow{\Gamma} \vdash \overleftrightarrow{\Delta}, \alpha{:}\overleftrightarrow{A}}\,(Ax)$$

$P \equiv \widehat{x}Q\widehat{\alpha}{\cdot}\beta :\ \overleftrightarrow{P} = [\widehat{x}\,\overleftrightarrow{Q}\,\widehat{\alpha}, \widehat{z}\langle y{\cdot}\delta \rangle \widehat{\delta}]{\cdot}\gamma$*. The 'iff' output circuit can only be typed with $(\leftrightarrow R)$, i.e.,*

$$\frac{Q :\cdot x{:}A, \Gamma \vdash_{\leftrightarrow} \Delta, \alpha{:}B \quad \langle y{\cdot}\delta \rangle :\cdot y{:}A, z{:}B, \Gamma \vdash_{\leftrightarrow} \Delta, \delta{:}A}{[\widehat{x}\,\overleftrightarrow{Q}\,\widehat{\alpha}, \widehat{z}\langle y{\cdot}\delta \rangle \widehat{\delta}]{\cdot}\gamma :\cdot y{:}A, \Gamma \vdash_{\leftrightarrow} \Delta, \gamma{:}A \leftrightarrow B}\,(\leftrightarrow R)$$

By induction we have, $Q :\cdot x{:}\overleftrightarrow{A}, \overleftrightarrow{\Gamma} \vdash \overleftrightarrow{\Delta}, \alpha{:}\overleftrightarrow{B}$, and we can construct the following derivation for P:

$$\frac{Q :\cdot x{:}\overleftrightarrow{A}, \overleftrightarrow{\Gamma} \vdash \overleftrightarrow{\Delta}, \alpha{:}\overleftrightarrow{B}}{\widehat{x}Q\widehat{\alpha}{\cdot}\beta :\cdot y{:}\overleftrightarrow{A}, \overleftrightarrow{\Gamma} \vdash \overleftrightarrow{\Delta}, \beta{:}\overleftrightarrow{A} \rightarrow \overleftrightarrow{B}}\,(\rightarrow R)$$

Note that the $y{:}A$ is not used in the construction of the type $\overleftrightarrow{A} \rightarrow \overleftrightarrow{B}$ (nor any type in $\vdash$), and can therefore be regarded as a weakened formula that is redundant in the proof.

$P \equiv M\widehat{\alpha}\,[z]\,\widehat{x}N :\ \overleftrightarrow{P} = y{\cdot}[\overleftrightarrow{M}\,\widehat{\alpha}\widehat{\beta}, \widehat{z}\widehat{x}\,\overleftrightarrow{N}]$ *can only be typed with the rule $(\leftrightarrow L)$,*

i.e.,

$$\frac{\overleftrightarrow{M} :\cdot \Gamma \vdash_{\leftrightarrow} \Delta, \alpha{:}A, \beta{:}B \quad \overleftrightarrow{N} :\cdot z{:}A, x{:}B, \Gamma \vdash_{\leftrightarrow} \Delta}{y{\cdot}[\overleftrightarrow{M}\widehat{\alpha}\widehat{\beta}, \widehat{z}\widehat{x}\overleftrightarrow{N}] :\cdot y{:}A \leftrightarrow B, \Gamma \vdash_{\leftrightarrow} \Delta} (\leftrightarrow L)$$

By induction twice, we have,

$$\begin{array}{c} M :\cdot \overleftrightarrow{\Gamma} \vdash \overleftrightarrow{\Delta}, \alpha{:}\overleftrightarrow{A}, \beta{:}\overleftrightarrow{B} \\ N :\cdot x{:}\overleftrightarrow{A}, y{:}\overleftrightarrow{B}, \overleftrightarrow{\Gamma} \vdash \overleftrightarrow{\Delta} \end{array}$$

Now we can construct the following derivation for P:

$$\frac{M :\cdot \overleftrightarrow{\Gamma} \vdash \overleftrightarrow{\Delta}, \alpha{:}\overleftrightarrow{A}, \beta{:}\overleftrightarrow{B} \quad N :\cdot x{:}\overleftrightarrow{A}, y{:}\overleftrightarrow{B} \vdash \overleftrightarrow{\Delta}}{M\widehat{\alpha}\,[z]\,\widehat{x}N :\cdot z{:}\overleftrightarrow{A} \rightarrow \overleftrightarrow{B}, y{:}\overleftrightarrow{B}, \overleftrightarrow{\Gamma} \vdash \overleftrightarrow{\Delta}, \beta{:}\overleftrightarrow{B}} (\rightarrow L)$$

$P \equiv M\widehat{\alpha} \dagger \widehat{x}N :\ \overleftrightarrow{P} = \overleftrightarrow{M}\widehat{\alpha} \dagger \widehat{x}\overleftrightarrow{N}$ *can only be typed with the rule* (Cut)*, i.e.,*

$$\frac{\overleftrightarrow{M} :\cdot \Gamma \vdash_{\leftrightarrow} \Delta, \alpha{:}A \quad \overleftrightarrow{N} :\cdot x{:}A, \Gamma \vdash_{\leftrightarrow} \Delta}{\overleftrightarrow{M}\widehat{\alpha} \dagger \widehat{x}\overleftrightarrow{N} :\cdot \Gamma \vdash_{\leftrightarrow} \Delta} (Cut)$$

By induction twice, we have,

$$\begin{array}{c} M :\cdot \overleftrightarrow{\Gamma} \vdash \overleftrightarrow{\Delta}, \alpha{:}\overleftrightarrow{A} \\ N :\cdot x{:}\overleftrightarrow{A}, \overleftrightarrow{\Gamma} \vdash \overleftrightarrow{\Delta} \end{array}$$

So we can construct the following derivation for P:

$$\frac{M :\cdot \overleftrightarrow{\Gamma} \vdash \overleftrightarrow{\Delta}, \alpha{:}\overleftrightarrow{A} \quad N :\cdot x{:}\overleftrightarrow{A}, \overleftrightarrow{\Gamma} \vdash \overleftrightarrow{\Delta}}{M\widehat{\alpha} \dagger \widehat{x}N :\cdot \overleftrightarrow{\Gamma} \vdash \overleftrightarrow{\Delta}} (Cut)$$

□

To show that our encoding is sensible, we must also check that we can simulate the reductions of $\mathcal{X}$. As pointed out in Section 5.2.3, the mechanism provided by the propagation and renaming rules is generic to any $\mathcal{X}$-style term calculus; it performs the same basic task of pushing cuts through subterms and renaming connectors regardless of the syntax employed. To show that such rules are simulated is straightforward, and we therefore only concern ourselves with the $\mathcal{X}$-calculus rules $(\textit{exp-imp}_{cbn})$ and $(\textit{exp-imp}_{cbv})$.

The following reduction confirms that we can simulate the rule $(\textit{exp-imp}_{cbv})$. The $\mathcal{X}^{\leftrightarrow}$ calculus can be extended with rules for garbage collection and renaming

similar to those of Lemma 3.1.25.

$$\begin{aligned}
&\overleftarrow{(\widehat{x}P\widehat{\alpha}\cdot\gamma)\widehat{\gamma}\dagger\widehat{y}(M\widehat{\mu}\,[y]\,\widehat{j}N)}\\
=\ &\overleftarrow{(\widehat{x}P\widehat{\alpha}\cdot\gamma)}\widehat{\gamma}\dagger\widehat{y}\overleftarrow{(M\widehat{\mu}\,[y]\,\widehat{j}N)}\\
=\ &([\widehat{x}\overleftarrow{P}\widehat{\alpha},\widehat{z}\langle c\cdot\delta\rangle\widehat{\delta}]\cdot\gamma)\widehat{\gamma}\dagger\widehat{y}(y\cdot[\overleftarrow{M}\widehat{\mu}\widehat{\sigma},\widehat{i}\widehat{j}\overleftarrow{N}]) \qquad z,c,\delta,\sigma,i \textit{ fresh}
\end{aligned}$$

Applying the rule $(\leftrightarrow_1)$ gives,

$$((\overleftarrow{M}\widehat{\mu}\dagger\widehat{x}\overleftarrow{P})\widehat{\sigma}\dagger\widehat{k}\langle k\cdot\alpha\rangle)\widehat{\alpha}\dagger\widehat{z}(\langle z\cdot\pi\rangle\widehat{\pi}\dagger\widehat{j}(\langle c\cdot\delta\rangle\widehat{\delta}\dagger\widehat{i}\overleftarrow{N}))$$

Since σ, i are fresh, we can garbage collect the cuts $\widehat{\sigma}\dagger\widehat{k}$ and $\widehat{\delta}\dagger\widehat{i}$, by applying the rules $(act\text{-L})$, $(\not\sim gc)$, $(act\text{-R})$, $(\not\sim gc)$, giving:

$$\begin{array}{lll}
(\overleftarrow{M}\widehat{\mu}\dagger\widehat{x}\overleftarrow{P})\widehat{\alpha}\dagger\widehat{z}(\langle z\cdot\pi\rangle\widehat{\pi}\dagger\widehat{j}\overleftarrow{N}) & \rightarrow & (ren\text{-R})\\
(\overleftarrow{M}\widehat{\mu}\dagger\widehat{x}\overleftarrow{P})\widehat{\alpha}\dagger\widehat{z}\overleftarrow{(N\{z/j\})} & =_\alpha & \\
(\overleftarrow{M}\widehat{\mu}\dagger\widehat{x}\overleftarrow{P})\widehat{\alpha}\dagger\widehat{j}\overleftarrow{N} & = & \\
\overleftarrow{M\widehat{\mu}\dagger\widehat{x}(P\widehat{\alpha}\dagger\widehat{j}N)} & &
\end{array}$$

In fact, our encoding is *only* able to simulate the $(exp\text{-}imp_{cbv})$ rule; the differently-bracketed alternatives of this rule may not reduce to each other and also do not always share the same set of normal forms.

The principal reduction rule for 'iff' manipulates four sub-circuits, while the principal reduction rule for any pairing connective involves three. We encoded implication by choosing one of the four sub-circuits to be a suitable capsule. Since the 'iff'-terms bind many combinations of inputs and outputs, we can suitably restrict them to computationally express other pairing connectives in a similar way. We are able to do this for the logical connectives $\wedge$ and $\uparrow$ up to the same limitations as discussed above for implication. Additionally, this can be achieved for the negation connective without limitations.

While the 'iff' connective is unable to *logically express* the connectives $\rightarrow$, $\wedge$, $\uparrow$, $\neg$, we are able to simulate some significant computational behaviour (i.e., a reduction subsystem) of their corresponding term calculi. In a symmetrical manner, the $\otimes$ connective is able to simulate the computational behaviour for the dual pairing connectives $-$, $\vee$, $\downarrow$ and again for the connective $\neg$.

5.6 Chapter Summary

In this chapter, we reviewed and presented the sequent calculus origins of the type system for the $\mathcal{X}$-calculus. The $\mathcal{X}$-calculus is built from a classical logic whose sole primitive connective is implication. We gave a general 'recipe' for building other interesting 'Curry-Howard pairs' of calculi based on other logical connectives. In particular, we detailed how to mechanically derive term annotations for sequent calculus proofs and extract a term calculus with a reduction mechanism based on a local cut-elimination procedure. We remark that the $\mathcal{X}$-calculus can be derived in this way.

One of the key choices in building Curry-Howard pairs of calculi is deciding on which connectives one should base their logic. We studied the class of sixteen arity-two connectives, and related them based on equivalences that employed simple negation operations. We found the sixteen connectives formed five groups of related connectives. We then studied the effect of these negation operations on the inference rules and cut-elimination rules for the logical connectives. We concluded that once the inference rules and cut-elimination rules are known for a connective of a particular group, one can apply the negation operations to determine the form and cut-elimination rules for the other connectives.

We studied one of these five groups of connectives in detail: the group of connectives which exhibited 'pairing' like functionality. For the connectives in this group, we showed that the 'logical expressibility' of a connective did correspond to its 'computational expressibility'.

We noticed that another of the five groups of connectives (consisting of the if-and-only-if and exclusive-or connectives) were largely unexplored in the literature. We gave a first definition for the sequent-calculus style inference rules and cut-elimination rules for this connective, then extracted a Curry-Howard pair of calculi using our 'recipe'. We showed that the iff connective could computationally express connectives which it could not logically express.

Finally, throughout the chapter, we motivated our view that when studying computational calculi (derived from logical calculi in the Curry-Howard sense), it is necessary to consider permutations of the principal reduction rule.

Chapter 6

Generalising the $\mathcal{X}$-calculus

In Section 5.2, we detailed a mechanical procedure for constructing Curry-Howard pairs of calculi from logical connectives. When we applied the procedure to construct a pair of calculi based on the if-and-only-if connective, we found the right-hand side of the principal reduction rules were not presented in their simplest form. In fact, we showed that the simplest presentation of the connective could not be derived by considering equivalent formulations using well studied connectives. We remarked in [74] that *perhaps* the simplest presentation of a term calculus based on if-and-only-if could not be derived in a automatic fashion. In this section we will show our original remark was misguided.

In Section 2.2.5, we reviewed some works which automatically generated principal logical rules, but noted that they used brute-force techniques. In some preliminary experiments we found that the brute-force technique did not scale to connectives of higher arities (e.g., greater than 5).

In this chapter, we will study principal reduction rules in detail. We will formalise an exact relationship between truth tables and sequent calculus inference rules for classical logical connectives, then using this relationship, present an algorithm which can intelligently (i.e., not using brute-force techniques) enumerate all 'good' principal reduction rules, enabling us to build an $\mathcal{X}$-style term calculus built from any classical logical connective.

6.1 Relating Truth-Tables and Inference Rules

In [24], Call informally describes a mechanical procedure that constructs a pair of invertible sequent calculus inference rules for a logical connective defined by

a truth table. In the following subsection we will formalise Call's work and give some intuitions we found that relate truth tables to sequent calculus rules. Developing on these intuitions, we are able to construct a reverse algorithm which constructs a truth table from a pair of inference rules (for a classical logical connective). This reverse algorithm is actually based on a 3-valued logic and gives some insight into how the cut rule operates on truth tables. Recall that our aim is to build a right-hand side of a principal logical rule; the structure of this derivation scheme is built using only applications of the cut rule, so a good understanding of its exact operation is important.

Since we aim to be as general as possible (within the scope of Classical Logic), we will first define the general shape or 'scheme' of a classical logical connective's inference rules. Recall that a scheme is an abstraction over inference rules (see Definition 2.2.4).

For each connective of true arity n there is an associated pair of invertible inference rules that introduce a formula with principal connective $\complement_i^n$ (for $0 \leq i < n$) and components $A_1, \ldots, A_n$ on the left- and right- hand sides of the rule conclusion.

We generalise the logical inference rules for classical propositional connectives by defining a notion of logical *rule-scheme* whose instances define a concrete pair of invertible inference rules for a particular connective.

Definition 6.1.1 (Inference Rule Schema) *A left and right logical* rule scheme *for an arbitrary connective $\complement_i^n$ has s and t many schemes for rule premises respectively (for $s \geq 0$ and $t \geq 0$). These 'sequent scheme' schemes serve to identify the formula and context variable parts of each premise that vary across the inference rules for each unique connective. A 'rule premise' scheme is then of the form $\Sigma, \Xi \vdash \Theta, \Lambda$ where:*

- Σ, Λ *are sets of formula schemes.*
- Ξ, Θ *are context metavariables.*

A 'rule conclusion' scheme is either of the form $\circledcirc, \Xi \vdash \Theta$ or of the form $\Xi \vdash \Theta, \circledcirc$, where the symbol $\circledcirc$ is a placeholder for a principal formula.

All rule-scheme variables for formula and context variables will be annotated with a superscript symbol either L or R to associate the variable with a left or right rule scheme respectively. The pair of left and right of rule schemes are then of the form:

$$\frac{\Sigma_1^L, \Xi_1^L \vdash \Theta_1^L, \Lambda_1^L \qquad \Sigma_2^L, \Xi_2^L \vdash \Theta_2^L, \Lambda_2^L \qquad \dots \qquad \Sigma_s^L, \Xi_s^L \vdash \Theta_s^L, \Lambda_s^L}{\varnothing, \bigsqcup_{k=1}^{s} \Xi_k^L \vdash \bigsqcup_{k=1}^{s} \Theta_k^L} (\mathsf{C}_i^n L)$$

$$\frac{\Sigma_1^R, \Xi_1^R \vdash \Theta_1^R, \Lambda_1^R \qquad \Sigma_2^R, \Xi_2^R \vdash \Theta_2^R, \Lambda_2^R \qquad \dots \qquad \Sigma_t^R, \Xi_t^R \vdash \Theta_t^R, \Lambda_t^R}{\bigsqcup_{k=1}^{t} \Xi_k^R \vdash \bigsqcup_{k=1}^{t} \Theta_k^R, \varnothing} (\mathsf{C}_i^n R)$$

An inference rule is then an instantiation of a rule scheme, with the following parameters supplied:

- *n, the arity of the connective.*
- *i, the unique connective (see Definition 2.2.11).*
- *s, the number of premises in the left rule.*
- *t, the number of premises in the right rule.*
- *Σ_i^L and Λ_i^L (for $0 < i \leq s$), the set of formula schemes for each left rule premise.*
- *Σ_i^R and Λ_i^R (for $0 < i \leq t$), the set of formula schemes for each right rule premise.*

Additionally, the scheme variables Ξ and Θ are promoted to context variables Γ and Δ.

Example 6.1.2 (An Instantiation of the Rule Scheme) *The inference rules for the connective $\mathsf{C}^2_{1101_2}$ (corresponding to implication) is obtained from the rule-scheme via the following instantiation.*

- *$n = 2$, $i = 1101_2$, $s = 2$, $t = 1$,*
- *$\Sigma_1^L = \varnothing$, $\Lambda_2^L = \{A_1\}$*
- *$\Sigma_2^L = \varnothing$, $\Lambda_2^L = \{A_2\}$*
- *$\Sigma_1^R = \{A_1\}$, $\Lambda_1^R = \{A_2\}$*
- *All scheme variables Ξ, Θ are promoted to context variables Γ, Δ.*

Then, the pair of invertible inference rules are:

$$\frac{\Gamma_1^L \vdash \Delta_1^L, A_1 \qquad A_2, \Gamma_2^L \vdash \Delta_2^L}{\mathsf{C}^2_{1101_2}(A_1, A_2), \Gamma_1^L, \Gamma_2^L \vdash \Delta_1^L, \Delta_2^L} (\mathsf{C}^2_{1101_2} L)$$

$$\frac{A_1, \Gamma_1^R \vdash \Delta_1^R, A_2}{\Gamma_1^L, \Gamma_2^L \vdash \Delta_1^L, \Delta_2^L, \mathsf{C}^2_{1101_2}(A_1, A_2)} (\mathsf{C}^2_{1101_2} R)$$

An algorithm to mechanically compute the sets Σ, Λ from the appropriate truth table will be given in Definition 6.1.6.

We place some constraints on the form of the inference rules described by the scheme so that only inference rules for classical logical connectives can be constructed; this introduces a notion of well-formedness.

Definition 6.1.3 (Well-Formed Logical Inference Rule) *We will place two restrictions on inference rule schemes in order to ensure the well-formedness of the left- and right- introduction rules for each arbitrary logical connective* $\complement_i^n(A_1, \ldots, A_1)$*. The restrictions are as follows:*

1. *No sub-component of the connective may appear on both sides of any particular rule premise, but each premise mentions at least one sub-component.*
   - *For each* $i \in \{1, \ldots, s\}$*:* $(\Sigma_i^L \cap \Lambda_i^L = \emptyset)$ *and* $(\Sigma_i^L \cup \Lambda_i^L \neq \emptyset)$
   - *For each* $i \in \{1, \ldots, t\}$*:* $(\Sigma_i^R \cap \Lambda_i^R = \emptyset)$ *and* $(\Sigma_i^R \cup \Lambda_i^R \neq \emptyset)$.
2. *Every argument or 'component' of the connective appears on the left of the turnstile of some rule premise* and *on the right of the turnstile of some (other) sequent scheme.*

$$\left(\bigcup_{i=1}^{s}\Sigma_i^L\right) \cup \left(\bigcup_{j=1}^{t}\Sigma_j^R\right) = \{A_1, \ldots, A_n\} \text{ and } \left(\bigcup_{i=1}^{s}\Lambda_i^L\right) \cup \left(\bigcup_{j=1}^{t}\Lambda_j^R\right) = \{A_1, \ldots, A_n\}$$

Observe that these restrictions capture the law of non-contradiction and the law of excluded middle for Classical Logic.

Definition 6.1.4 (Linear representation of Well-Formed Inference Rules) *An inference rule formalises the definition of a connective (as discussed in Section 2.2). By taking the classical interpretation of a sequent* $\Gamma \vdash \Delta$*, interpreting the comma's on the left and right of the sequent as conjunction and disjunction respectively, we can express an inference rule in a (linear) propositional language built from propositional variables* x, x, x *and propositional connectives (in descending order of binding strength)* $!, \&\&, ||, \Rightarrow, \top\!\!\top, \bot\!\!\bot$*. These symbols denote the usual notions of negation, conjunction, disjunction, implication, truth and falsehood. In this language, the general form of a left inference rule (i.e., the left rule scheme) can be written as:*

$$\mathop{\&\&}_{i=1}^{s}\left(\Xi_i^L \&\& \Sigma_i^L \Rightarrow \Lambda_i^L || \Theta_i^L\right) \Rightarrow \left(\left(\mathop{\&\&}_{i=1}^{s}\Xi_i^L \&\& \complement_i^n(A_1, \ldots, A_n)\right) \Rightarrow \Big|\Big|_{i=1}^{s}\Theta_i^L\right) \tag{6.1}$$

and the general form of a right inference rule (the right rule scheme) can be written as:

$$\mathop{\&\&}_{i=1}^{t}\left(\Xi_i^R \&\& \Sigma_i^R \Rightarrow \Lambda_i^R || \Theta_i^R\right) \Rightarrow \left(\left(\mathop{\&\&}_{i=1}^{t}\Xi_i^R\right) \Rightarrow \left(\Big|\Big|_{i=1}^{t}\Theta_i^R || \complement_i^n(A_1, \ldots, A_n)\right)\right) \tag{6.2}$$

Note that this 'flattening' of the rule scheme is a purely syntactic transformation which can be reversed (i.e, the syntax used in the above representations can be rearranged and the previous notation adopted).

We introduce a notion of principal reduction rule on sequent calculus proofs that exactly follows cut-elimination.

6.1.1 The Principal Reduction Rule Scheme

We will adopt terminology from rewriting when speaking about cut-elimination transformations. A cut-elimination transformation is a rewrite rule defined on a set of proofs. We shall write *left-hand side* and *right-hand side* when we mean to refer to the left and right-hand sides of a cut-elimination rule respectively. An instantiation of a left-hand side and right-hand side will be referred to as a redex and contractum respectively.

The application of a principal reduction rule to a proof will eliminate an instance of a connective $\complement_i^n(A_1, \ldots, A_n)$ from the proof when it has been immediately introduced as the cut-formula by the inference rules $(\complement_i^n R)$ (with t premises) and $(\complement_i^n L)$ (with s premises). The general form or *scheme* of the rule's left-hand side is shown below, with $0<(s+t)\leq 2^n$, $0<p\leq s$ and $0<q\leq t$.

$$\dfrac{\dfrac{\ldots \quad \overbrace{\Sigma_q^R, \Gamma_q^R \vdash \Delta_q^R, \Lambda_q^R}^{D_{Rq}} \quad \ldots}{\bigcup_{k=1}^{t} \Gamma_k^R \vdash \bigcup_{k=1}^{t} \Delta_k^R, \complement_i^n(A_1, \ldots, A_n)}(\complement_i^n R) \qquad \dfrac{\ldots \quad \overbrace{\Sigma_p^L, \Gamma_p^L \vdash \Delta_p^L, \Lambda_p^L}^{D_{Lp}} \quad \ldots}{\complement_i^n(A_1, \ldots, A_n), \bigcup_{k=1}^{s} \Gamma_k^L \vdash \bigcup_{k=1}^{s} \Delta_k^L}(\complement_i^n L)}{\bigcup_{k=1}^{s} \Gamma_k^L \bigcup_{k=1}^{t} \Gamma_k^R \vdash \bigcup_{k=1}^{s} \Delta_k^L \bigcup_{k=1}^{t} \Delta_k^R}(Cut)$$

Where D_{Lp} and D_{Rq} are variables over derivation schemes ending in the sequent scheme $\Sigma_p^L, \Gamma_p^L \vdash \Delta_p^L, \Lambda_p^L$ and $\Sigma_q^R, \Gamma_q^R \vdash \Delta_q^R, \Lambda_q^R$ respectively called *proof variables*. These *proof variables* are reminiscent of the metavariables used in term rewriting.

The right-hand side of the principal reduction rule represents the proof scheme that is the result of removing the above cut from the derivation. The form of the right-hand side is a derivation scheme of the same endsequent, except it is derived from only one or more applications of the cut rule and the leaves of the derivation schemes are proof variables.

We will use the term *principal reduction rule* to refer to an instantiation of a principal reduction rule scheme.

6.1.2 Formalising Call's Algorithm

Call's algorithm, which builds a pair of sequent calculus inference rules for a classical logical connective from a truth table, has two steps:

Step One : the extraction of the premises of the inference rule from the truth table.

Step Two : the simplification of the extracted set of premises.

We will illustrate the relationship between truth tables and sequent calculus inference rules with an example, before giving the formal definition.

Example 6.1.5 (Extracting Inference Rules for Implication via equivalences) *The truth function for logical implication is given by the following truth table.*

	A_1	A_2	$A_1 \rightarrow A_2$
0	0	0	1
1	0	1	1
2	1	0	0
3	1	1	1

We can extract semantics for the above truth table using the language given in Definition 6.1.4. We obtain the following pair of expressions from the above truth table.

$$!(A_1 \rightarrow A_2) \Rightarrow A_1 \,\&\&\, !A_2 \tag{6.3}$$

$$A_1 \rightarrow A_2 \Rightarrow (!A_1 \,\&\&\, !A_2) \;||\; (!A_1 \,\&\&\, A_2) \;||\; (A_1 \,\&\&\, A_2) \tag{6.4}$$

The above equations can be rewritten in the form of equations (6.1) *and* (6.2) *from Definition 6.1.4 using the classical equivalences* ($A_1 || A_2 \equiv !(!A_1 \,\&\&\, !A_2)$, $!(A_1 \,\&\&\, A_2) \equiv (A_1 \Rightarrow !A_2)$, $A_1 \Rightarrow A_2 \equiv (!A_2 \Rightarrow !A_1)$, $A_1 \equiv \top\!\!\top \Rightarrow A_1$ *and* $!A_1 \equiv \bot\!\!\bot \Rightarrow A_1$*) as shown*

below.

$$
\begin{aligned}
(6.3) &= !(A_1 \rightarrow A_2) \Rightarrow A_1 \,\&\&\, !A_2 \\
&\equiv !(A_1 \rightarrow A_2) \Rightarrow !(A_1 \Rightarrow A_2) \\
&\equiv (A_1 \Rightarrow A_2) \Rightarrow A_1 \rightarrow A_2
\end{aligned}
$$

$$
\begin{aligned}
(6.4) &= A_1 \rightarrow A_2 \Rightarrow (!A_1 \,\&\&\, !A_2) \;||\; (!A_1 \,\&\&\, A_2) \;||\; (A_1 \,\&\&\, A_2) \\
&\equiv A_1 \rightarrow A_2 \Rightarrow !(\; !(!A_1 \,\&\&\, !A_2) \;\&\&\; !(!A_1 \,\&\&\, A_2) \;\&\&\; !(A_1 \,\&\&\, A_2) \;) \\
&\equiv A_1 \rightarrow A_2 \Rightarrow !(\; (!A_1 \Rightarrow A_2) \;\&\&\; (!A_1 \Rightarrow !A_2) \;\&\&\; (A_1 \Rightarrow !A_2) \;) \\
&\equiv (\; (!A_1 \Rightarrow A_2) \;\&\&\; (!A_1 \Rightarrow !A_2) \;\&\&\; (A_1 \Rightarrow !A_2) \;) \Rightarrow !(A_1 \rightarrow A_2) \\
&\equiv (\; (\top\!\!\top \Rightarrow A_1 || A_2) \;\&\&\; (A_2 \Rightarrow A_1) \;\&\&\; (A_1 \,\&\&\, A_2 \Rightarrow \bot\!\!\!\bot) \;) \Rightarrow (A_1 \rightarrow A_2) \Rightarrow \bot\!\!\!\bot
\end{aligned}
$$

The equivalences above translate the propositional formulas extracted from the truth table to a form comparable to the linear representation of rule schemes (Definition 6.1.4). This process is optimised by the following algorithm.

Definition 6.1.6 (Extracting Inference Rules) *A connective $\complement_i^n(A_1, \ldots, A_n)$ is defined by a truth function. Recall (from Definition 2.2.11) that we defined the truth function as $\mathbb{C}_i^n :: [\mathcal{T}] \rightarrow \mathcal{T}$. We associate with each row indexed r of the truth table either the sequent $\Sigma_r^L, \Xi_r^L \vdash \Theta_r^L, \Lambda_r^L$ or the sequent $\Sigma_r^R, \Xi_r^R \vdash \Theta_r^R, \Lambda_r^R$ as follows:*

$$
\begin{aligned}
\Sigma_r^L &= \left[A_c \mid \mathbb{C}_i^n(\ddot{n}_r) = 1 \wedge 0{<}c{\leq}n \wedge \mathbb{C}_i^n[r][c] = 1\right] \\
\Lambda_r^L &= \left[A_c \mid \mathbb{C}_i^n(\ddot{n}_r) = 1 \wedge 0{<}c{\leq}n \wedge \mathbb{C}_i^n[r][c] = 0\right]
\end{aligned}
$$

$$
\begin{aligned}
\Sigma_r^R &= \left[A_c \mid \mathbb{C}_i^n(\ddot{n}_r) = 0 \wedge 0{<}c{\leq}n \wedge \mathbb{C}_i^n[r][c] = 1\right] \\
\Lambda_r^R &= \left[A_c \mid \mathbb{C}_i^n(\ddot{n}_r) = 0 \wedge 0{<}c{\leq}n \wedge \mathbb{C}_i^n[r][c] = 0\right]
\end{aligned}
$$

The above sets can be used to instantiate a rule scheme. The parameters n and i are available from the truth table itself, while the parameters s and t are a count of the number of 1's and 0's respectively in the defining column of the truth table.

We give an example illustrating the use of the above algorithm.

Example 6.1.7 (Extracting Inference Rules for Implication via algorithm) *Applying the algorithm of Definition 6.1.6 to the truth table for implication in Example 6.1.5, we*

obtain the following inference rules (with some renumbering of indexes),

$$\frac{\Gamma_1^L \vdash \Delta_1^L, A_1, A_2 \qquad A_2, \Gamma_2^L \vdash \Delta_2^L, A_1 \qquad A_1, A_2, \Gamma_3^L \vdash \Delta_3^L}{\mathtt{C}^2_{1101_2}(A_1, A_2), \Gamma_1^L, \Gamma_2^L, \Gamma_3^L \vdash \Delta_1^L, \Delta_2^L, \Delta_3^L}\ (\mathtt{C}^2_{1101_2}L)$$

$$\frac{A_1, \Gamma_1^R \vdash \Delta_1^R, A_2}{\Gamma_1^R \vdash \Delta_1^R, \mathtt{C}^2_{1101_2}(A_1, A_2)}\ (\mathtt{C}^2_{1101_2}R)$$

The left introduction rule above does not correspond to the usual left introduction rule, which employs only two premises, since it is not yet in its 'simplest form'.

Call noticed that step one of the algorithm did not always build the simplest form of inference rules. In fact, the rules generated by step one will only be in their simplest form when the connective is a *parity connective* (i.e., a connective that cannot be shortcut like for example exclusive-or or if-and-only-if); such connectives require the values for all of their arguments to be known before computation.

In the example above, there are redundancies within the set of premises belonging to the left introduction rule. Specifically, the cut rule can be applied to some pairs of premises. Step two of Call's algorithm exhaustively applies the cut rule to pairs of premises within a particular rule. The result is a simplified set of premises, which Call uses to build the simplest form of a rule. In addition to the data structures defined in the notation section (Section 2.1), in this chapter we add the following, which will be used to model inference rules and principal reduction rules.

Definition 6.1.8 (Formulas, Sequent Schemes and Derivation Schemes) *The algorithms we will specify in this chapter work on inference rules (rather than proofs). We define the following three datatypes for use in our algorithms.*

Formula Variable *: An element of type* `formula` *corresponds to a 'formula variable' that appears in an inference rule.*

Sequent Scheme *: Making the distinction between proof sequents and the sequent schemes used in inference rules, we introduce the datatype* `sscheme` *(which represents a sequent scheme). Elements of* `sscheme` *are pairs (similar to sequents), and each component of the pair is a set of* `formulas`*. However, in our discussions we may use the words sequent and sequent scheme interchangeably when there is no possibility of confusion.*

Derivation Scheme *: An element of* `dscheme` *(pronounced 'derivation scheme') is a scheme for building a concrete proof, and has the structure of a tree. Derivation schemes are recursively defined as:*

$$\texttt{dscheme} ::= \texttt{sscheme} \times [\texttt{dscheme}]$$

(where of course the empty-list is used to construct the base case). In the following, we will abbreviate a derivation scheme $\langle(\Sigma \vdash \Lambda), D_i\rangle$ *to* $\mathcal{D}_i$*.*

Using the above structures, we formally define step two of Call's algorithm below.

Definition 6.1.9 (Simplification of Rule Premises) *The simplification procedure applies the cut rule to pairs of premises that mention the same set of formulas in their sequents, and share a unique cut-formula. We specify the* `simplify` *procedure as,*

$$\begin{array}{l}
\texttt{cancut} :: \texttt{dscheme} \to \texttt{dscheme} \to \texttt{boolean} \\
\texttt{cancut}\ \langle(\Theta_1 \vdash \Theta_2), L\rangle\ \langle(\Theta_3 \vdash \Theta_4), R\rangle = (\Theta_2 \cap \Theta_3 = \{A\}) \wedge (\Theta_1 \cup \Theta_2 = \Theta_3 \cup \Theta_4) \\
\\
\texttt{cut} :: \texttt{dscheme} \to \texttt{dscheme} \to \texttt{formula} \to \texttt{dscheme} \\
\texttt{cut}\ \langle(\Theta_1 \vdash \Theta_2), L\rangle\ \langle(\Theta_3 \vdash \Theta_4), R\rangle\ cf = \langle\ (\Theta_1 \cup (\Theta_3 \setminus cf) \vdash (\Theta_2 \setminus cf) \cup \Theta_4), \\
\qquad\qquad\qquad\qquad\qquad\qquad [\langle(\Theta_1 \vdash \Theta_2), L\rangle, \langle(\Theta_3 \vdash \Theta_4), R\rangle]\ \rangle \\
\\
\texttt{simplify} :: \{\texttt{dscheme}\} \to \{\texttt{dscheme}\} \\
\texttt{simplify}\ P \mid (P = P') = P \\
\qquad\qquad\quad \mid \texttt{otherwise} = \texttt{simplify}\ P' \\
\qquad\qquad\quad\ \texttt{where}\ P' = (P \cup new) \setminus old \\
\qquad\qquad\qquad\qquad new = \{\langle z, D\rangle \mid L, R \in P \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \texttt{cancut}\ L\ R \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \langle(\Theta_1 \vdash \Theta_2), X\rangle = L \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \langle(\Theta_3 \vdash \Theta_4), Y\rangle = R \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \langle z, Z\rangle = \texttt{cut}\ L\ R\ (\Theta_2 \cap \Theta_3) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ D \textit{ is a fresh proof variable } \} \\
\qquad\qquad\qquad\qquad old = \{L, R \mid L, R \in P \wedge \texttt{cancut}\ L\ R\}
\end{array}$$

We will apply the above definition to the inference rules generated in Example 6.1.7.

Example 6.1.10 (Minimisation of Premises for Implication) *We apply* `simplify` *to (i) the set of left premises, then (ii) to the set of right premises of the inference rules*

for implication that were generated in Example 6.1.7. We will at times omit the context variables Γ, Δ to improve readability.

(i) $\texttt{simplify}\ \{\langle(\vdash A_1, A_2), D_{L1}\rangle, \langle(A_2 \vdash A_1), D_{L2}\rangle, \langle(A_1, A_2 \vdash), D_{L3}\rangle\}$
$= (\{\langle(\vdash A_1, A_2), D_{L1}\rangle, \langle(A_2 \vdash A_1), D_{L2}\rangle, \langle(A_1, A_2 \vdash), D_{L3}\rangle\} \cup$
$\{\langle(\vdash A_1), D'_{L1}\rangle, \langle(A_2 \vdash), D'_{L2}\rangle\}) \setminus$
$\{\langle(\vdash A_1, A_2), D_{L1}\rangle, \langle(A_2 \vdash A_1), D_{L2}\rangle, \langle(A_1, A_2 \vdash), D_{L3}\rangle\}$
$= \{\langle(\vdash A_1), D'_{L1}\rangle, \langle(A_2 \vdash), D'_{L2}\rangle\}$

(ii) $\texttt{simplify}\ \{\langle(A_1 \vdash A_2), D_{R1}\rangle\} = \{\langle(A_1 \vdash A_2), D_{R1}\rangle\}$

Using these reduced premises, we can instantiate the rule scheme with $n = 2$, $i = 1101_2$, $s = 2$, $t = 1$, giving:

$$\frac{\Gamma_1^L \vdash \Delta_1^L, A_1 \quad A_2, \Gamma_2^L \vdash \Delta_2^L}{\mathsf{C}^2_{1101_2}(A_1, A_2), \Gamma_1^L, \Gamma_2^L \vdash \Delta_1^L, \Delta_2^L}\,(\mathsf{C}^2_{1101_2}L) \qquad \frac{A_1, \Gamma_1^R \vdash \Delta_1^R, A_2}{\Gamma_1^R \vdash \Delta_1^R, \mathsf{C}^2_{1101_2}(A_1, A_2)}\,(\mathsf{C}^2_{1101_2}R)$$

Notice that these are exactly the pair of inference rules for implication.

6.1.3 Truth Tables from Inference Rules

In this section, we will design an algorithm `buildmask`, which computes the reverse process of Call's algorithm, i.e., one which associates rows of a truth table with a sequent.

We will begin by examining the effect of applying the `cut` procedure to the rule premises for the implication connective (as used by `simplify` in Example 6.1.10). In the following discussion, we will omit context variables Γ, Δ to improve readability.

Looking back at Example 6.1.10, step one of Call's Algorithm makes the following associations between the rows of a truth table and the premises of the inference rules for implication.

	A_1	A_2	$\mathsf{C}^2_{1101_2}(A_1, A_2)$		Corresponding Sequent
0	0	0	1	$\approx$	$\vdash A_1, A_2$
1	0	1	1	$\approx$	$A_2 \vdash A_1$
2	1	0	0	$\approx$	$A_1 \vdash A_2$
3	1	1	1	$\approx$	$A_1, A_2 \vdash$

Step two involved applying the procedure cut to the of the left implication rule, which produced the simplified sequent schemes $(\quad \vdash A_1)$ and $(A_2 \vdash \quad)$, i.e.,

$$\texttt{cut}\ \langle(\quad \vdash A_1, A_2), D_{L1}\rangle\ \langle(A_2 \vdash A_1), D_{L2}\rangle\ A_2 = \langle(\quad \vdash A_1), D'_{L1}\rangle \tag{6.5}$$

$$\texttt{cut}\ \langle(A_2 \vdash A_1), D_{L2}\rangle\ \langle(A_1, A_2 \vdash \quad), D_{L3}\rangle\ A_1 = \langle(A_2 \vdash \quad), D'_{L2}\rangle \tag{6.6}$$

In sequent calculus form, we would write:

$$\frac{\vdash A_1, A_2 \quad A_2 \vdash A_1}{\vdash A_1}\,(Cut) \qquad\qquad \frac{A_1 \vdash A_2 \quad A_1, A_2 \vdash}{A_2 \vdash}\,(Cut)$$

Recall that the truth value of a formula $\complement_i^n(A_1, \ldots, A_n)$ is determined by its inputs (i.e., the truth values of its arguments), and also, that the truth table definition of a connective enumerates all possible inputs for the connective. For a truth table $\mathbb{C}_i^n$, the first n elements of row r (represented as the length-n list $\ddot{n}_r$) uniquely describes one possible input, i.e., the list $\ddot{n}_r$ describes the case when connective's arguments $A_1, \ldots, A_n$ has truth values $\ddot{n}_r[0], \ldots, \ddot{n}_r[n-1]$.

Since each premise of a left rule is associated with a row where the connective is assigned a truth value of 1, applying the (Cut) rule to a pair of left premises can be seen as removing information not required to compute that value of 1. Take the first application of (Cut) shown above; its two premises are extracted from rows 0 and 1 of the above truth table. In both rows, the truth value of A_1 is 0 (A_2 has different truth values). This means that whenever A_1 has value 0, the value of the connective is 1, regardless of the value of A_2. Allowing for some rearrangement of the corresponding sequent scheme, this observation can be seen by considering the equivalence: $(A_1 \vee A_2) \wedge (A_1 \vee \neg A_2) \equiv A_1$.

The same argument can be applied to the second application of (Cut) shown above. The two premises of this cut are extracted from rows 1 and 3 of the truth table. Both rows describe inputs where A_2 has a truth value of 1 (and different truth values for A_1). This means that whenever A_2 has the truth value 1, the value of the connective is immediately known to be 1 (again, regardless of the value of A_1).

With these observations in mind, it is reasonable to associate the sequent $(\quad \vdash A_1)$ with two rows of the truth table: rows 0 *and* 1: these two rows correspond to all the cases where A_1 has truth value 0, and A_2 has different truth values. In this case we will say that we *don't care* about the truth value given to A_2. Similarly, it is reasonable to associate $(A_2 \vdash \quad)$ with two rows 1 and 3, since these rows correspond to inputs where A_2 has the truth value 1. In this case, we *don't care*

about the truth value of A_1. We associate the premise $(A_1 \vdash A_2)$ with only row 2, since this was the original association, and the procedure `cut` was not applied to this premise.

We can make some generalisations of the above associations. Suppose, starting from the truth table definition of $\complement_i^n(A_1, \ldots, A_n)$, we extract two sets of rule premises (to build the left and right rules) using Call's algorithm. We can associate a set of rows with each rule premise according to the following specification, where each row is represented as a length-n list (denoting a binary number).

- If the formula A_i appears on the left of the turnstile, each input in the set of associated rows has truth value 1 at position i of the list.
- If the formula A_i appears on the right of the turnstile, each input in the set of associated rows has the truth value 0 at position i of the list.
- If the formula A_i does not appear on either side of the sequent, then we 'didn't care' about the truth value of A_i when computing the truth value of the connective. In this case, we enumerate all possible assignments of truth values to A_i, and include these in the set of inputs to correspond to the sequent. For example, notice for the set of inputs $\{\dddot{2}_0, \dddot{2}_1\} = \{[0,0],[0,1]\}$ associated with the sequent $(\quad \vdash A_1)$, all possible truth values for the argument A_2 are considered.

We remark that no formula will appear on both sides of the sequent, since such an inference rule would not be well-formed (according to Definition 6.1.3).

We formalise the above notions in Definition 6.1.12, but first we will define a convenient structure which we call a *bitmask* (and is related to three-valued truth assignments) that we will use to relate sequent schemes to sets of truth function inputs.

Definition 6.1.11 (Bitmask) *An* input *is an element in the domain of a truth function. A* bitmask *is a set of inputs. Each input in the bitmask is represented as a list of length n consisting of elements from $\mathcal{T}^X = \mathcal{T} \cup \{X\}$, where X denotes the* don't care *state, i.e,*

$$\texttt{bitmask} = [\mathcal{T}^X]$$

We can explicitly represent set of row indices of the bitmask by `expand`*ing it as follows:*

$$\begin{aligned}&\texttt{expand} :: \texttt{bitmask} \to \{\texttt{int}\}\\ &\texttt{expand}\ \mathit{mask} = \left\{ i \mid (0 \leq i < 2^{|\mathit{mask}|}) \wedge \texttt{fit}\ \mathit{mask}\ (|\dddot{\mathit{mask}}|_i) \right\}\end{aligned}$$

Notice that if b is a length-n bitmask with has k 'don't care' elements, `expand` *b is a set of 2^k integers.*

We will say an input is in a bitmask if it `fit`*s the bitmask, where* `fit` *is a predicate defined as follows:*

$$\begin{aligned}
&\texttt{fit} :: [\mathcal{T}] \to [\mathcal{T}^X] \to \texttt{boolean} \\
&\texttt{fit}\ [\,]\ [\,] \qquad\quad = \mathit{true} \\
&\texttt{fit}\ r{:}\mathit{row}\ m{:}\mathit{mask} = ((r{=}m) \vee (m{=}X)) \wedge \texttt{fit}\ \mathit{row}\ \mathit{mask}
\end{aligned}$$

In the following we will write bitmasks simply as $b_1 b_2 \ldots b_n$ (instead of $[b_1, b_2, \ldots, b_n]$), where $b_i \in \mathcal{T}^X$ and $0{<}i{\leq}n$.

Using bitmasks, we can more easily specify an algorithm to compute a set of truth table rows that corresponds to a sequent.

Definition 6.1.12 (Build Mask, `buildmask`**)** *Given a sequent scheme S (generated by applying Call's algorithm to a truth table $\mathbb{C}_i^n$), the procedure call* `buildmask` *S n computes a set of inputs with which S can be associates. The procedure* `buildmask` *is defined below.*

$$\begin{aligned}
&\texttt{buildmask}\ :: \texttt{sscheme} \to \texttt{int} \to \texttt{bitmask} \\
&\texttt{buildmask}\ S\ i \mid (i = 0) = [\,] \\
&\qquad\qquad\quad \mid \texttt{otherwise} = (\texttt{buildmask}\ S\ (i{-}1)) {+}{+} [\texttt{valueof}\ S\ i] \\
&\qquad\qquad\quad \texttt{where valueof}\ (\Theta_1 \vdash \Theta_2)\ k \mid (A_k \in \Theta_1 \wedge A_k \notin \Theta_2) = 0 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid (A_k \notin \Theta_1 \wedge A_k \in \Theta_2) = 1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid (A_k \notin \Theta_1 \wedge A_k \notin \Theta_2) = X
\end{aligned}$$

Example 6.1.13 (Bitmasks for the Rules of the Implication Connective) *Take the simplified inference rules for implication generated in Example 6.1.10. We apply the procedure* `buildmask` *to each premise of the left and right inference rule (also passing the arity of the connective) as follows.*

$$\begin{aligned}
\texttt{buildmask}\ (\quad \vdash A_1)\ 2 &= 0X \\
\texttt{buildmask}\ (A_2 \vdash \quad)\ 2 &= X1 \\
\texttt{buildmask}\ (A_1 \vdash A_2)\ 2 &= 10
\end{aligned}$$

Notice that,

$$\begin{aligned}
\texttt{expand}\ 0X &= \{00_2, 01_2\} = \{0, 1\} \\
\texttt{expand}\ X1 &= \{01_2, 11_2\} = \{1, 3\} \\
\texttt{expand}\ 10 &= \{10_2\} \qquad = \{2\}
\end{aligned}$$

Also,

$$\begin{array}{ll}
\texttt{fit}\ \dddot{2}_0\ 0X = \texttt{fit}\ [0,0]\ 0X = \textit{true} & \texttt{fit}\ \dddot{2}_0\ X1 = \texttt{fit}\ [0,0]\ X1 = \textit{false} \\
\texttt{fit}\ \dddot{2}_1\ 0X = \texttt{fit}\ [0,1]\ 0X = \textit{true} & \texttt{fit}\ \dddot{2}_1\ X1 = \texttt{fit}\ [0,1]\ X1 = \textit{true} \\
\texttt{fit}\ \dddot{2}_2\ 0X = \texttt{fit}\ [1,0]\ 0X = \textit{false} & \texttt{fit}\ \dddot{2}_2\ X1 = \texttt{fit}\ [1,0]\ X1 = \textit{false} \\
\texttt{fit}\ \dddot{2}_3\ 0X = \texttt{fit}\ [1,1]\ 0X = \textit{false} & \texttt{fit}\ \dddot{2}_3\ X1 = \texttt{fit}\ [1,1]\ X1 = \textit{true}
\end{array}$$

In fact, we can also rebuild the sequent scheme from the bitmask by supplying the bitmask to the following procedure.

Definition 6.1.14 (Build Sequent, `buildsequent`) *We can (re)construct a sequent from a bitmask by applying the procedure* `buildsequent` *to the bitmask, where:*

$$\begin{aligned}
&\texttt{buildsequent} :: \texttt{bitmask} \longrightarrow \texttt{sscheme} \\
&\texttt{buildsequent}\ m = \texttt{bs}'\ m\ 1\ (\emptyset \vdash \emptyset)
\end{aligned}$$

where,

$$\begin{aligned}
&\texttt{bs}' :: \texttt{bitmask} \longrightarrow \texttt{int} \longrightarrow \texttt{sscheme} \longrightarrow \texttt{sscheme} \\
&\texttt{bs}'\ [\,]\ k\ (\Theta_1 \vdash \Theta_2) \qquad\quad = \Theta_1 \vdash \Theta_2 \\
&\texttt{bs}'\ m{:}\textit{mask}\ k\ (\Theta_1 \vdash \Theta_2) \ \mid\ (m = 0) = \texttt{bs}'\ \textit{mask}\ (k{+}1)\ (\Theta_1 \vdash \Theta_2 \cup \{A_k\}) \\
&\qquad\qquad\qquad\qquad\quad \mid\ (m = 1) = \texttt{bs}'\ \textit{mask}\ (k{+}1)\ (\{A_k\} \cup \Theta_1 \vdash \Theta_2) \\
&\qquad\qquad\qquad\qquad\quad \mid\ (m = X) = \texttt{bs}'\ \textit{mask}\ (k{+}1)\ (\Theta_1 \vdash \Theta_2)
\end{aligned}$$

For convenience, we will introduce some terminology which directly relates rows of the truth table with a sequent scheme generated by Call's algorithm.

Definition 6.1.15 (Covers) *We will say a sequent scheme S (generated by applying Call's algorithm to the truth table $\mathbb{C}_i^n$)* covers *the set of rows R of the truth table, where:*

$$R = \texttt{expand}\ (\texttt{buildmask}\ S\ n)$$

We will sometimes overload our use of the term covers *by applying it directly to bitmasks, (as in a bitmask* covers *the rows R).*

6.2 Applying the Cut Rule to Truth Tables

In this section, we will overload the procedure `cut` to operate directly on bitmasks. This is actually straightforward, since in the previous section, we speci-

fied how to obtain a bitmask from a sequent (using the procedure `buildmask`) and also how to obtain a sequent from a bitmask (using the procedure `buildsequent`). However, we will leave the formal definition to the end of this section, since a deeper analysis of the operation will reveal more insight into the cut rule.

Ultimately, our goal is to automate the construction of the principal reduction rule for a logical connective. As we observed in Section 5.5 when studying the $\mathcal{X}^{\leftrightarrow}$-calculus, there may be several ways of building the rule—some being more complicated than others. Initially, we considered constructing an algorithm which would find *all* principal reduction rules for the connective. This algorithm used a brute force search as would be adopted by the resolution based algorithms of Ciabattoni and Leitsch [30] and Baaz *et al.* [5] (except their algorithms sought a *single* principal reduction rule, for their motives for seeking the rule were different). Our brute force algorithm resembled the following:

$$
\begin{aligned}
&\texttt{bruteforce} :: \{\texttt{dscheme}\} \to \{\texttt{dscheme}\} \\
&\texttt{bruteforce}\ P \ \mid (P = P') = P \\
&\qquad\qquad\quad \mid \texttt{otherwise}\ = \texttt{bruteforce}\ P' \\
&\qquad\qquad\quad\ \ \texttt{where}\ P' = P \cup \{\texttt{cut}\ x\ y \ \mid\ x, y \in P\}
\end{aligned}
$$

Even with considerable pruning of the search space (while still attempting to compute the set of 'all' principal reduction rules), the above algorithm spent much time building duplicate results, and, more worryingly building sequent schemes of the form $A_1, \ldots, A_n \vdash A_1, \ldots, A_n$. We will show later that such schemes, where a formula appears on both sides of the turnstile, are undesired. Additionally, for certain connectives, the brute-force procedure attempted the construction of a set of infinitely many derivations schemes, and therefore failed to terminate. Considering the simplest example of this case, we can see that the conclusion of a cut whose premises are $A \vdash B$ and $B \vdash A$ is again applicable to one of those premises *ad infinitum*. The existence of such arbitrary rules led us to formulate the notion of a 'good' principal reduction rule. Recall that when we studied principal reduction rules for the $\mathcal{X}^{\leftrightarrow}$-calculus, we were able to formulate two pairs of rules: a pair that involved making copies of some rule premises (Definition 5.5.1), and a 'simplified' pair without copying (Definition 5.5.2). We will say a principal reduction rule is *good* if the right-hand side does not duplicate any proof variable (i.e., rule premises). In term rewriting terminology, we would say that we require that the principal reduction rule to be *right linear*. 'Good' rules will of course exclude infinite derivations.

When seeking right-hand sides of principal reduction rules, we will not want to

take a brute-force approach. This means we require a notion of progress when building a right-hand side (by applying the cut rule to various combinations of derivation schemes built from proof variables). A vague notion can already be formulated by examining an instance of a reduction rule. Consider the proof transformation that was used to derive the principal reduction rule $(\leftrightarrow_1)$ of the $\mathcal{X}^{\leftrightarrow}$-calculus (see Definition 5.5.2).

$$\dfrac{\dfrac{\boxed{D_{R1}}\quad\boxed{D_{R2}}}{\Gamma_1^L \vdash \Delta_1^L, A_1, A_2 \qquad A_2, A_1, \Gamma_2^L \vdash \Delta_2^L}(\leftrightarrow R) \qquad \dfrac{\boxed{D_{L1}}\quad\boxed{D_{L2}}}{A_1, \Gamma_1^R \vdash \Delta_1^R, A_2 \qquad A_2, \Gamma_2^R \vdash \Delta_2^R, A_1}(\leftrightarrow L)}{\Gamma_1^L, \Gamma_2^L \vdash \Delta_1^L, \Delta_2^L, A_1 \leftrightarrow A_2 \qquad A_1 \leftrightarrow A_2, \Gamma_1^R, \Gamma_2^R \vdash \Delta_1^R, \Delta_2^R}(Cut)}{\Gamma_1^L, \Gamma_2^L, \Gamma_1^R, \Gamma_2^R \vdash \Delta_1^L, \Delta_2^L, \Delta_1^R, \Delta_2^R}$$

$$\downarrow$$

$$\dfrac{\dfrac{\dfrac{\boxed{D_{R1}}\quad\boxed{D_{L1}}}{\Gamma_1^L \vdash \Delta_1^L, A_1, A_2 \qquad A_1, \Gamma_1^R \vdash \Delta_1^R, A_2}(Cut) \qquad \dfrac{\boxed{D_{L2}}\quad\boxed{D_{R2}}}{A_2, \Gamma_2^R \vdash \Delta_2^R, A_1 \qquad A_2, A_1, \Gamma_2^L \vdash \Delta_2^L}(Cut)}{\Gamma_1^L, \Gamma_1^R \vdash \Delta_1^L, \Delta_1^R, A_2 \qquad A_2, \Gamma_2^L, \Gamma_2^R \vdash \Delta_2^L, \Delta_2^R}(Cut)}{\Gamma_1^L, \Gamma_2^L, \Gamma_1^R, \Gamma_2^R \vdash \Delta_1^L, \Delta_2^L, \Delta_1^R, \Delta_2^R}$$

The sequent schemes at the leaves of a principal reduction rule have the most formula variables. In the reduction rule above, A_1 and A_2 both appear in *every* leaf. The sequent scheme at the root of the derivation mentions no formula variable. The cut rule reduces the number of formula variables in each sequent scheme at successive levels of the derivation (remember our rules have implicit contraction). The constructed right-hand side use the cut to eliminate all components of the connective from the sequent schemes (leaving only the context variables).

Using bitmasks rather than sequent schemes gives an alternate (and simpler) notion of progress. Consider the above right-hand side constructed from bitmask representations of sequent schemes:

$$\dfrac{\dfrac{\dfrac{\boxed{D_{R1}}\quad\boxed{D_{L1}}}{00 \qquad 10}(Cut) \qquad \dfrac{\boxed{D_{L2}}\quad\boxed{D_{R2}}}{01 \qquad 11}(Cut)}{X0 \qquad X1}(Cut)}{XX}$$

Notice that the bitmask XX covers every row of the truth table. This makes sense since the right-hand side of the principal reduction rule should be a complete encoding of the truth function denoted by the connective. Notice that each application of the cut rule sets the truth value of one element of the bitmask to 'don't care', and, by the time the derivation scheme is constructed, the truth values of all elements of the bitmask are 'don't care'.

The principal reduction rules for $\mathcal{X}^{\leftrightarrow}$ are based on a fairly simple connective of only arity two. We will apply our findings to the more complicated connective, $\mathsf{C}^3_{00001001_2}$ (abbreviated to C^3_9), which is defined by the following truth function.

Example 6.2.1 (The Connective C^3_9) *The connective C^3_9 behaves like the bottom connective whenever the truth value of the first argument is false, and otherwise behaves as the if-and-only-if connective.*

The truth table $\mathbb{C}^3_9$ (defining the connective) is shown below, together with the inference rules $(\mathsf{C}^3_9 L)$ and $(\mathsf{C}^3_9 R)$ generated by applying Call's algorithm to the truth table. (We have annotated each sequent scheme with the corresponding bitmask).

	A_1	A_2	A_3	$\mathsf{C}^3_9(A_1, A_2, A_3)$
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	0
7	1	1	1	1

$$\dfrac{\begin{array}{cc}\underset{100}{\boxed{D_{L1}}} & \underset{111}{\boxed{D_{L2}}} \\ A_1 \vdash A_2, A_3 & A_1, A_2, A_3 \vdash\end{array}}{\mathsf{C}^3_9(A_1, A_2, A_3) \vdash}\,(\mathsf{C}^3_9 L)$$

$$\dfrac{\begin{array}{ccc}\underset{0XX}{\boxed{D_{R1}}} & \underset{X01}{\boxed{D_{R2}}} & \underset{X10}{\boxed{D_{R3}}} \\ \vdash A_1 & A_3 \vdash A_2 & A_2 \vdash A_3\end{array}}{\vdash \mathsf{C}^3_9(A_1, A_2, A_3)}\,(\mathsf{C}^3_9 R)$$

Like the $\mathcal{X}^{\leftrightarrow}$ case, it is not immediately obvious how to begin building a principal reduction rule for the above connective (not to mention trying to build the set of all 'good' principal reduction rules). We will give a case analysis of applications of the cut rule to the premises of the rules $(\mathsf{C}^3_9 L)$ and $(\mathsf{C}^3_9 R)$, and focus on bitmasks in our discussions rather than sequent schemes. The interesting cases are illustrated in Figure 6.1.

Remember that the procedure we aim to define in this section is,

$$\texttt{cut} :: \texttt{bitmask} \to \texttt{bitmask} \to \texttt{bitmask}$$

In Figure 6.1(a), the cut rule is not applicable to the pair of premises, i.e., there is no suitable cut formula. Comparing the bitmasks, the truth value of *every* formula is 'don't care' in at least one of the two premises (i.e., the premises have no formulas in common). There is a second (trivial) case when the cut rule cannot be applied to a pair of bitmasks; consider a cut between D_{R1} and itself: the two bitmasks will have the same truth value for each formula.

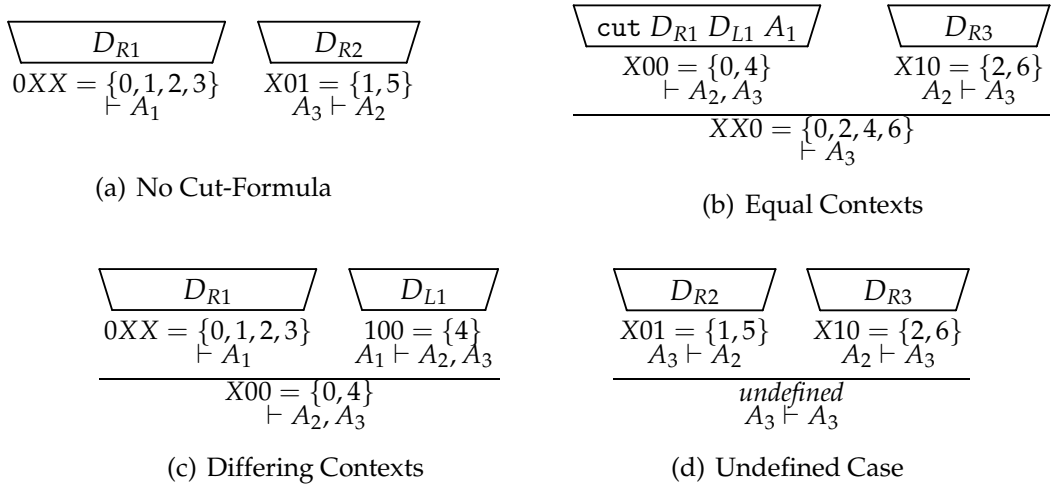


Figure 6.1: Applications of (Cut) to premises of the rules $(\complement_9^3 L)$ and $(\complement_9^3 R)$.

In Figure 6.1(b), the bitmasks agree on the formulas they 'don't care' about, and disagree on the truth value of exactly one formula. This case highlights an ideal application of the cut rule. Note that the ordering of premises is important in the cut rule: the cut formula should be in the succedent of the left premise and in the antecedent of the right premise. Analogously, the truth value of the 'cut formula element' in the left and right bitmasks are 1 and 0 respectively. Notice that in this ideal case, the conclusion of the cut covers all the rows covered by its premises (i.e., the cut performs a union of the sets of rows covered by the premises).

In Figure 6.1(c), the cut rule is applicable where the cut formula is A_1 (i.e., the first element of the left and right bitmasks are 0 and 1 respectively). In this case, notice that the rows covered by the conclusion of the cut are only a selective union of the rows covered its premises. This can be explained by first noticing that the contexts of the premises differ. We are working with a multiplicative formulation of the cut rule, and so the contexts are merged in the conclusion of the cut (hence the formulas A_2 and A_3 being in the conclusion's succedent despite not being in the left premise's succedent). With bitmasks, disagreeing contexts are identified where one bitmask 'doesn't care' about the truth value of a particular formula, while the other 'does' (i.e., has truth value from $\mathcal{T}$). The context merging operation is encoded in the conclusion's bitmask by inheriting the known truth value for the 'don't care' formula from the opposite premise. It is this context merging which determines which rows are 'selected' when the cut takes the union of the rows covered by its premises. This is seen much easier using an additive formulation of the cut rule (where the respective contexts of the two premises of the cut must agree on all formulas, with the exception of the cut formula). Using an explicit weakening rule, we could successively add the

relevant formulas to the cut's left premise as follows:

$$\dfrac{\dfrac{\dfrac{\boxed{D_{R1}}}{0XX = \{0,1,2,3\} \vdash A_1}}{00X = \{0,1\} \vdash A_2, A_1}\,(Weakening_R)}{000 = \{0\} \vdash A_2, A_3, A_1}\,(Weakening_R) \qquad \dfrac{\boxed{D_{L1}}}{100 = \{4\} \quad A_1 \vdash A_2, A_3}}{X00 = \{0,4\} \vdash A_2, A_3}\,(Cut)$$

This shows that the left premise, when placed in a cut with the right premise, in fact only covered one row of the truth table. An additive formulation of the cut rule always performs a direct union of rows covered by a sequent scheme. Also notice that after weakening, the bitmasks for the premises of the cut are of the ideal form as illustrated in Figure 6.1(b): they agree on the truth values of all formulas except for the truth value of the formula which is 0 in the left bitmask and 1 in the right bitmask. Observe that the weakened formula is chosen from the right premise, or, the truth value of each 'don't care' is inherited from the bitmask on the right.

In Figure 6.1(d), the bitmasks disagree on the truth value of a formula which is not the cut formula (i.e., A_3). Since only the cut formula can be eliminated from the pair of sequent schemes, the multiplicative cut rule will copy the formula A_3 to both sides of the turnstile in the conclusion. Recall that we associate formulas on the left (right) of the sequent scheme with truth table rows where the input to the connective has truth value 1 (0). A formula cannot have two truth values (by the classical law of non-contradiction), and so there is no corresponding bitmask for this case.

Finally, we summarise the above discussion in the following definition.

Definition 6.2.2 (Cut Applied to Bitmasks) *Given two bitmasks b_1, b_2 of equal length, we define a procedure* `cut`*, which computes the bitmask corresponding to the conclusion of a cut with premises* `buildsequent` b_1 *and* `buildsequent` b_2 *respectively.*

$$\begin{aligned}
&\texttt{cut} :: \texttt{bitmask} \to \texttt{bitmask} \to \texttt{bitmask} \\
&\texttt{cut}\ [\,]\ [\,] &&= [\,] \\
&\texttt{cut}\ \mathit{l}{:}\mathit{lmask}\ \mathit{r}{:}\mathit{rmask} &&= (\texttt{resolve}\ \mathit{l}\ \mathit{r}) : \texttt{cut}\ \mathit{lmask}\ \mathit{rmask} \\
&&&\quad \texttt{where}\ \texttt{resolve}\ 0\ 1 = X \\
&&&\qquad\quad \texttt{resolve}\ X\ t = t \\
&&&\qquad\quad \texttt{resolve}\ t\ X = t
\end{aligned}$$

In order to ensure the resulting bitmask is not 'undefined', we need a precondition on the rule which ensures the two bitmasks disagree on the truth value of exactly one formula. That is, we require `disagreecount` $= 1$, *where the function* `disagreecount` *is defined as:*

$$
\begin{aligned}
&\texttt{disagreecount} :: \texttt{bitmask} \to \texttt{bitmask} \to \texttt{int} \\
&\texttt{disagreecount}\ [\,]\ [\,] \qquad\qquad = 0 \\
&\texttt{disagreecount}\ \mathit{l}\texttt{:}\mathit{lmask}\ \mathit{r}\texttt{:}\mathit{rmask} = (\texttt{disagree}\ \mathit{l}\ \mathit{r}) + \texttt{disagreecount}\ \mathit{lmask}\ \mathit{rmask} \\
&\qquad\qquad\qquad\qquad\qquad\quad \texttt{where disagree}\ t\ t\ \ = 0 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{disagree}\ X\ t = 0 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{disagree}\ t\ X = 0 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{disagree}\ 1\ 0\ = 1
\end{aligned}
$$

Now we can see that our task of building a right-hand side can be equated to building a bitmask where every element has been set to 'don't care' (by successively applying the `cut` procedure to pairs of bitmasks that correspond to the premises of a pair of inference rules).

6.3 On the Geometry of Principal Reduction Rules

In this section we will take a slight digression and (informally) present a geometrical analogy for a pair of sequent calculus inference rules for a logical connective. This view is not essential to the understanding of our main algorithm, but we discuss it here since it may give the reader some extra insight into the operations performed by bitmasks.

If we enumerate the inputs of an arbitrary truth function $\mathbb{C}_i^n$ and represent each input as a length-n bitmask (with zero 'don't care' elements), we can apply the `cut` procedure (from Definition 6.2.2) to a pair of those bitmasks exactly when they differ in a single truth value (i.e., have `disagreecount` $= 1$). In the field of information theory, a metric called the *Hamming distance* is calculated as a count of the number of positions that differ between two strings of symbols of equal length. Applying this metric to our context, the 'strings' are bitmasks, and the symbols are elements of $\mathcal{T}^X$. We can apply `cut` to any pair of bitmasks that have a Hamming distance of 1.

There is a convenient geometrical structure called a *hypercube graph* that is often used to calculate the Hamming distance between two strings. As an example, we

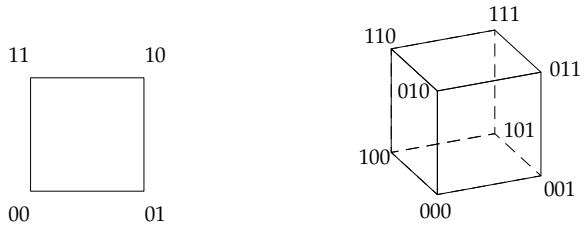


Figure 6.2: The Hamming 2-Cube and 3-Cube

show the 2-dimensional and 3-dimensional hypercubes (more commonly called 'square' and 'cube') in Figure 6.2. We have labelled each node of the hypercube with a unique input to the truth function, and arranged the node labels so that a *line* joins pairs of bitmasks that have a Hamming distance of 1. (Note that we will use the word *line* exclusively to refer to the *skeleton* of the n-cube which joins pairs of nodes having Hamming distances equal to 1). Under this configuration, the lines represent all possible applications of `cut` to those sets of bitmasks. To encode a connective's truth function, we must graphically associate (a representation for) a truth value with each input. We can do this by assigning a colour to each vertex. We colour the nodes of the n-cube for the truth function $\mathbb{C}_i^n$ as follows.

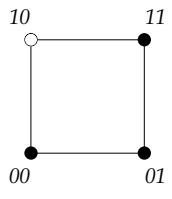
- A node is coloured *white* if the label of the node corresponds to an input of the truth function where the connective evaluates to *false*.
- A node is coloured *black* if the label of the node corresponds to an input of the truth function where the connective evaluates to *true*.

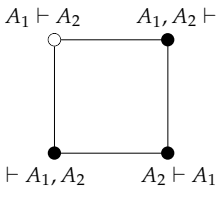
It follows directly that the *unsimplified* inference rules for a logical connective are also encoded within the coloured hypercube (i.e., the pair of inference rules obtained by applying only step one of Call's algorithm to $\mathbb{C}_i^n$). One can observe this explicitly by (i) replacing every node label b, with the sequent scheme obtained by applying Definition 6.1.6 to b, (ii) assigning every node coloured black as a premise of the left introduction rule and (iii) assigning every node coloured white as a premise of the right introduction rule.

We give some examples below.

Example 6.3.1 (Hypercube for $\mathbb{C}_{1101_2}^2$) *The truth function $\mathbb{C}_{1101_2}^2$ is encoded in the following 2-cube.*

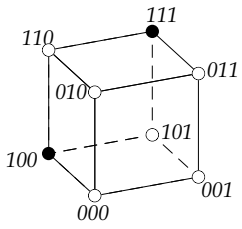
	A_1	A_2	$\mathbb{C}^2_{1101_2}(A_1, A_2)$
0	0	0	1
1	0	1	1
2	1	0	0
3	1	1	1

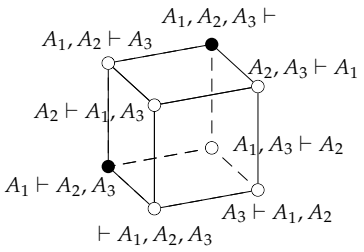




Example 6.3.2 (Hypercube for $\mathbb{C}^3_9$) *The truth function $\mathbb{C}^3_9$ is encoded in the following 3-cube.*

	A_1	A_2	A_3	$\mathbb{C}^3_9(A_1, A_2, A_3)$
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	0
7	1	1	1	1





Each row of the truth table is represented as a labelled coloured node. The colour of the node labelled with the bitmask b is black if $\mathbb{C}^3_9(r) = 1$, otherwise it is white.

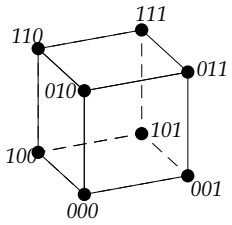
Having constructed an representation of the truth function for a connective as a hypercube (and also a representation of the unsimplified inference rules for a connective), we asked whether it was possible to encode the *simplified* representation of an inference rule within a hypercube. Such an encoding would give some insight into the cut operation when viewed as a geometrical operation on hypercubes.

Recalling that a simplified rule premise corresponds to a bitmask with a number of 'don't care' elements, we began by generalising our representation of bitmasks to include those with any number of 'don't care' elements.

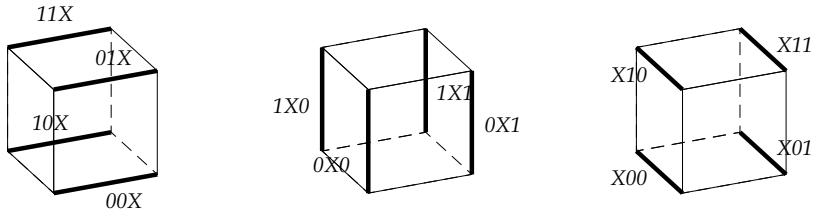
Our investigations revealed that we could encode a length-n bitmask with k 'don't care' elements $(0 \leq k \leq n)$ as a k-dimensional hypercube in an n-dimensional space. We illustrate this encoding for the arity three connectives below.

Example 6.3.3 (Encoding Bitmasks of Arity 3) *Take an arbitrary connective of arity 3. The bitmasks corresponding to sets of rows of the truth table $\mathbb{C}^3_i$ (with $0 \leq i < 2^{3^3}$) will be of length 3. Also, recall that if b is a bitmask with k 'don't care' elements,* `expand` *b is a set of 2^k bitmasks with zero 'don't care' elements. Now, considering all possible bitmasks of length three, we can make the following associations.*

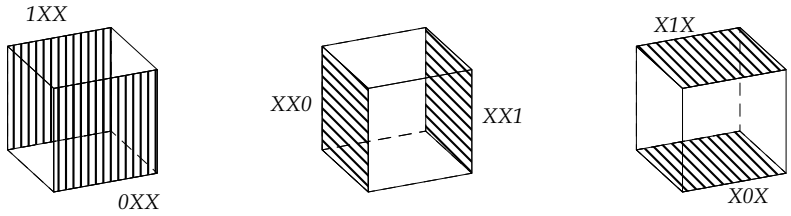
- *We can map each bitmask with* zero *'don't care' elements to a unique 0-dimensional hypercube (i.e., a vertex) on the 3-cube as follows:*



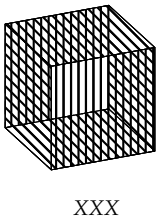
- *We can map each bitmask with* one *'don't care' element (representing a set of* 2^1 *bitmasks with zero 'don't care' elements) to a unique 1-dimensional hypercube (i.e., an edge) on the 3-cube as follows:*



- *We can map each bitmask with* two *'don't care' elements (representing a set of* 2^2 *bitmasks with zero 'don't care' elements) to a unique 2-dimensional hypercube (i.e., a face) on the 3-cube as follows:*



- *We can map the bitmask with* three *'don't care' elements (representing a set of* 2^3 *bitmasks with zero 'don't care' elements) to a 3-dimensional hypercube as follows:*



Now we will consider the effect of the procedure `cut` on a pair of hypercubes.

Applying `cut` to a pair of 'appropriate' bitmasks (i.e., a pair bitmasks that have a `disagreecount` $= 1$ and k 'don't care' elements at common indexes) produces a

bitmask with $(k+1)$ 'don't care' elements. Recall that the task of building a right-hand side can be expressed in the language of bitmasks: a right-hand side is built by applying `cut` to pairs of bitmasks in some order so that the final application builds the bitmask with all elements set to 'don't care'.

In our geometrical setting, we observe that applying `cut` to an 'appropriate' pair of k-cubes builds a $(k+1)$-cube. The task of building a right-hand side (for the connective $\complement_i^n$) is that of building an n-cube in an n-dimensional space by successively applying `cut` to 'merge' appropriate pairs of k-cubes until the n-cube is produced. We illustrate this notion with a simple example.

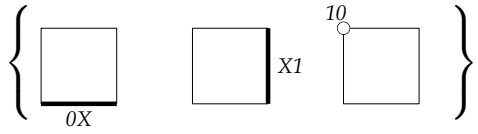
Example 6.3.4 (Building the Right-hand Side For Implication) *The set of* simplified *premises for implication are:*

$$\{(\quad \vdash A_1), (A_2 \vdash \quad), (A_1 \vdash A_2)\}$$

Expressed as bitmasks, this is equal to:

$$\{0X, X1, 10\}$$

Projecting the hypercubes onto the 'skeleton' of a 2-cube, we have:



The principal reduction rules for implication are well-known and are straightforward to build. We could build the bitmask XX from the above set of premise bitmasks (or *corresponding sequent schemes) with the following applications of* `cut`*:*

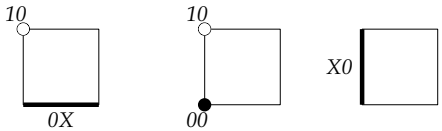
$$\begin{aligned} &\texttt{cut}\ (\texttt{cut}\ 0X\ 10)\ X1 \\ &= \texttt{cut}\ (\texttt{cut}\ 00\ 10)\ X1 \\ &= \texttt{cut}\ X0\ X1 \\ &= XX \end{aligned}$$

or

$$\begin{aligned} &\texttt{cut}\ (\texttt{cut}\ (\quad \vdash A_1)\ (A_1 \vdash A_2)\ )\ (A_2 \vdash \quad) \\ &= \texttt{cut}\ (\texttt{cut}\ (\quad \vdash A_1, A_2)\ (A_1 \vdash A_2)\ )\ (A_2 \vdash \quad) \\ &= \texttt{cut}\ (\vdash A_2)\ (A_2 \vdash \quad) \\ &= (\quad \vdash \quad) \end{aligned}$$

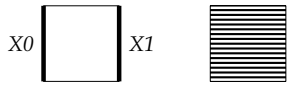
We can mirror this process on the set of corresponding hypercubes. In the following, we superimpose the hypercubes that represent the arguments of the `cut` *procedure onto a single 2-cube.*

$$\texttt{cut}\ 0X\ 10 = \texttt{cut}\ 00\ 10 = \quad X0$$



Followed by,

$$\texttt{cut}\ X0\ X1 = \quad XX$$



Notice that in the first application of `cut`*, the 1-cube labelled* $0X$ *was first* `resolved` *(Definition 6.2.2) to the 0-cube labelled* 00*. (This corresponds to the resolving of contexts we discussed in Section 6.2).*

The application of `cut` *to the two edges* $X0$ *and* $X1$ *produces the 2-cube. Observe that in this case, every point on one edge is within a Hamming distance of 1 with some point on the other edge.*

The general case of applying the `cut` to a pair of hypercubes is slightly complicated. Suppose we have two hypercubes: (i) an i-cube $cube_1$ and (ii) a j-cube $cube_2$ (with $0 \leq i \leq j \leq n$). The application,

$$\texttt{cut}\ cube_1\ cube_2$$

proceeds as follows:

1. $cube_1$ is replaced with a p-cube $cube'_1$ with $(0 \leq p \leq i)$ such that every point on $cube'_1$ is within a Hamming distance of 1 of some point on $cube_2$.
2. $cube_2$ is replaced with a q-cube $cube'_2$ with $(0 \leq q \leq j)$ such that every point on $cube'_2$ is within a Hamming distance of 1 of some point on $cube_1$.
3. If every point on $cube'_1$ is within a Hamming distance of 1 of some point on $cube'_2$, then the precondition `disagreecount` $= 1$ held; notice in this case $p = q$. Otherwise `cut` is not applicable to $cube_1$ and $cube_2$.
4. The result $cube_3$ is the $(p+q)$-cube consisting of all points on $cube'_1$ and $cube'_2$.

We will look at a more involved example that illustrates the above steps at the end of the next section.

6.4 Enumerating Principal Reduction Rules

Let us recap the progress we have made so far in this chapter and review our main goal. Our aim is to automate the building of Curry-Howard calculi for Classical Logic. The difficulty in automating a local cut-elimination procedure lies in building the 'good' principal logical rule for a logical connective. Using

them method of considering logical equivalences does not always build the simplest form of the rule (as we observed in Section 5.5), and so another method is needed. We have spent the first sections of this chapter formulating precisely a criteria for 'progress' when building a right-hand side of a principal logical rule (for an arbitrary connective $\complement_i^n$). This criteria has in fact been specified using three different analogies, which we reiterate below.

1. To build a derivation scheme from applications of (Cut) whose endsequent or 'root' contains only context variables and whose leaves are premises of the rules $(\complement_i^n L)$ and $(\complement_i^n R)$.
2. To build a bitmask $X_1 \ldots X_n$ from applications of `cut` to pairs of bitmasks corresponding to the premises of the rules $(\complement_i^n L)$ and $(\complement_i^n R)$.
3. To build an n-cube in an n-dimensional space by successively 'merging' i- and j-cubes ($0 \leq i \leq j \leq n$).

In this section we will address the *order* in which the bitmasks should be cut, and therefore the order in which the (Cut) rule should be applied to premises to build the right-hand side of a principal reduction rule.

Working backwards, we could construct the right-hand side of a principal reduction rule for a connective $\complement_i^n$ from the root of the derivation scheme rather than from the leaves. First observe that the root is built by an application of some *topmost cut* (since only applications of the cut rule are allowed to build the right-hand side). This topmost cut will eliminate the final component of the connective from the derivation scheme, and will therefore be of the following shape (omitting context variables):

$$\dfrac{\begin{array}{cc} \boxed{D_1} & \boxed{D_2} \\ X_1 \ldots X_{i-1} 0 X_{i+1} \ldots X_n & X_1 \ldots X_{i-1} 1 X_{i+1} \ldots X_n \\ \vdash A_i & A_i \vdash \end{array}}{\begin{array}{c} X_1 \ldots X_n \\ \vdash \end{array}} (Cut)$$

Where $0 < i \leq n$.

That is, the topmost cut must be made between two bitmasks that disagree on the value of exactly one formula *and* the truth value of all other formulas is 'don't care'. Observe that the left and right bitmasks in a topmost cut will each cover 2^{n-1} rows of the truth table. Essentially, we have split the problem of building the root bitmask into two sub-problems. Since the cut rule performs only a selective[1]

[1]Recall that the sets represented by the two bitmasks are first resolved on 'don't care' states corresponding to weakening steps that merge contexts in the conclusion of the derivation scheme.

union operation on rows covered by bitmasks (i.e., the conclusion cannot cover any rows which were not covered by the premises), we can split the rule premises into two sets: those which cover the rows covered by $(\quad \vdash A_i)$ and those which cover the rows covered by $(A_i \vdash \quad)$. The following example illustrates this first step.

Example 6.4.1 (Topmost Cut for C_9^3) *The choices of the topmost cut of the right-hand side of a principal reduction rule for the C_9^3 connective are:*

$$\dfrac{\begin{array}{cc}\boxed{D_{R1}} & \boxed{D_1} \\ 0XX = \{0,1,2,3\} & 1XX = \{4,5,6,7\} \\ \vdash A_1 & A_1 \vdash\end{array}}{XXX = \{0,1,2,3,4,5,6,7\} \vdash}\,(Cut) \qquad \dfrac{\begin{array}{cc}\boxed{D_2} & \boxed{D_3} \\ X0X = \{0,1,4,5\} & X1X = \{2,3,6,7\} \\ \vdash A_2 & A_2 \vdash\end{array}}{XXX = \{0,1,2,3,4,5,6,7\} \vdash}\,(Cut)$$

$$\dfrac{\begin{array}{cc}\boxed{D_4} & \boxed{D_5} \\ XX0 = \{0,2,4,6\} & 1XX = \{1,3,5,7\} \\ \vdash A_3 & A_3 \vdash\end{array}}{XXX = \{0,1,2,3,4,5,6,7\} \vdash}\,(Cut)$$

Taking the first variant, we can group the premises of the rules $(\mathsf{C}_9^3 L)$ and $(\mathsf{C}_9^3 R)$ into two sets: those which will be used to build the sequent scheme $(\quad \vdash A_1)$, and those which will build $(A_1 \vdash \quad)$. Notice that the premise $\mathcal{D}_{R1}$ is already of the required form (i.e., its endsequent is $(\vdash A_1)$). The remaining premises (i.e., $\{\mathcal{D}_{L1}, \mathcal{D}_{R2}, \mathcal{D}_{R3}, \mathcal{D}_{L2}\}$) can be used to build a sequent scheme which covers the rows $\{4,5,6,7\}$ (since those premises cover those rows). This is can be seen directly if we build a map from each row of the truth table to the rule premises which cover that row.

Row	*Rule Premise covering Row*
0	$\{\mathcal{D}_{R1}\}$
1	$\{\mathcal{D}_{R1}, \mathcal{D}_{R2}\}$
2	$\{\mathcal{D}_{R1}, \mathcal{D}_{R3}\}$
3	$\{\mathcal{D}_{R1}\}$
4	$\{\mathcal{D}_{L1}\}$
5	$\{\mathcal{D}_{R2}\}$
6	$\{\mathcal{D}_{R3}\}$
7	$\{\mathcal{D}_{L2}\}$

We will define a refinement of the above structure which we call a base map *in the definition following this example (see Definition 6.4.2).*

Now we can see that the set of premises $\{\mathcal{D}_{L1}, \mathcal{D}_{R2}, \mathcal{D}_{R3}, \mathcal{D}_{L2}\}$ will construct the rows which cover $(A_1 \vdash \quad)$ since only those premises cover the rows $4, 5, 6$ and 7.

To complete the example, we observe that the left and right premises of the second variant of the topmost cut can be constructed from the respective sets $\{\mathcal{D}_{R1}, \mathcal{D}_{R2}, \mathcal{D}_{L1}\}$ and $\{\mathcal{D}_{R1}, \mathcal{D}_{R3}, \mathcal{D}_{L2}\}$. Similarly, the third variant can be constructed from $\{\mathcal{D}_{R1}, \mathcal{D}_{R3}, \mathcal{D}_{L1}\}$ and $\{\mathcal{D}_{R1}, \mathcal{D}_{R2}, \mathcal{D}_{L2}\}$. We briefly point out here that in both of these cases, the premise $\mathcal{D}_{R1}$ might appear on both sides of the derivation (scheme) tree since it appears in the sets that will be used to build both the left and right premises.

Definition 6.4.2 (Base Map) *Given a pair of inference rules $(\complement_i^n L)$ and $(\complement_i^n R)$ generated by applying Call's algorithm to the truth table $\mathbb{C}_i^n$, we define a* base map *as a mapping from each row of the truth table to a* single *rule premise. Since there may exist more than one such mapping (consider the case when more than one rule premise covers a particular row), we define a procedure* `baseMaps` *that computes the set of all such unique base maps.*

Starting from the set S of all rule premises of the rules $(\complement_i^n L)$ and $(\complement_i^n R)$, we construct the set of all pairs of 'rows and rule premises that cover the row' with the following function.

$$\begin{aligned}&\texttt{allbasepairs} :: \{\texttt{dscheme}\} \to \texttt{int} \to \{\texttt{int} \times \texttt{dscheme}\}\\ &\texttt{allbasepairs}\ S\ n = \{\langle i, d\rangle \mid d \in S \wedge i \in \texttt{expand}\ (\texttt{buildmask}\ d\ n)\}\end{aligned}$$

We group the 'base pairs' into n sets, so that each set contains only base pairs for a particular row. The generic `group` *function is defined below.*

$$\begin{aligned}&\texttt{group} :: \{a \times b\} \to \{\{a \times b\}\}\\ &\texttt{group}\ S = \{\ \{\langle i, \_\rangle \mid \langle i, \_\rangle \in S\} \mid i \in \{j \mid \langle j, \_\rangle \in S\}\}\end{aligned}$$

The cartesian product of these grouped sets of base pairs will construct all unique base maps.

$$\begin{aligned}&\texttt{baseMaps} :: \{\texttt{dscheme}\} \to \texttt{int} \to \{\texttt{int} \times \texttt{dscheme}\}\\ &\texttt{baseMaps}\ S\ n = \prod \texttt{group}\ (\texttt{allbasepairs}\ S\ n)\end{aligned}$$

We give some intuition behind the above structure with an example.

Example 6.4.3 (Base Map for $\complement_9^3$) *We combine the rule premises of the rules $(\complement_9^3 L)$ and*

($\complement_9^3 R$) *(see Example 6.2.1), and build the set of all premises S for the connective* $\complement_9^3$*:*

$$\begin{aligned} S = \{\ & \langle (A_1 \vdash A_2, A_3), D_{L1} \rangle, \\ & \langle (A_1, A_2, A_3 \vdash\ \ ), D_{L2} \rangle, \\ & \langle (\ \ \vdash A_1), D_{R1} \rangle, \\ & \langle (A_3 \vdash A_2), D_{R2} \rangle, \\ & \langle (A_2 \vdash A_3), D_{R3} \rangle\ \} \end{aligned}$$

We first build the set of all base pairs,

$$\begin{aligned} \texttt{allbasepairs}\ S\ 3 = \{\ & \langle 0, \mathcal{D}_{R1} \rangle,\ \langle 1, \mathcal{D}_{R1} \rangle,\ \langle 1, \mathcal{D}_{R2} \rangle,\ \langle 2, \mathcal{D}_{R1} \rangle,\ \langle 2, \mathcal{D}_{R3} \rangle, \\ & \langle 3, \mathcal{D}_{R1} \rangle,\ \langle 4, \mathcal{D}_{L1} \rangle,\ \langle 5, \mathcal{D}_{R2} \rangle,\ \langle 6, \mathcal{D}_{R3} \rangle,\ \langle 7, \mathcal{D}_{L2} \rangle\ \} \end{aligned}$$

The above set tells us row 0 is covered by the premise $\mathcal{D}_{R1}$*, row 1 is covered by the* $\mathcal{D}_{R1}$ *and* $\mathcal{D}_{R2}$*, row 2 is covered by* $\mathcal{D}_{R1}$ *and* $\mathcal{D}_{R3}$*, and so on. Grouping these base pairs on the row index, we get,*

$$\begin{aligned} \texttt{group}\ (\texttt{allbasepairs}\ S\ 3) = \{\ & \{\langle 0, \mathcal{D}_{R1} \rangle\},\ \{\langle 1, \mathcal{D}_{R1} \rangle, \langle 1, \mathcal{D}_{R2} \rangle\}, \\ & \{\langle 2, \mathcal{D}_{R1} \rangle, \langle 2, \mathcal{D}_{R3} \rangle\},\ \{\langle 3, \mathcal{D}_{R1} \rangle\}, \\ & \{\langle 4, \mathcal{D}_{L1} \rangle\},\ \{\langle 5, \mathcal{D}_{R2} \rangle\},\ \{\langle 6, \mathcal{D}_{R3} \rangle\}, \\ & \{\langle 7, \mathcal{D}_{L2} \rangle\}\ \} \end{aligned}$$

The cartesian product of the above set generates all possible base maps as follows,

$$\begin{aligned} & \prod \texttt{group}\ (\texttt{allbasepairs}\ S\ 3) \\ = \{\ & \{\langle 0, \mathcal{D}_{R1} \rangle, \langle 1, \mathcal{D}_{R1} \rangle, \langle 2, \mathcal{D}_{R1} \rangle, \langle 3, \mathcal{D}_{R1} \rangle, \langle 4, \mathcal{D}_{L1} \rangle, \langle 5, \mathcal{D}_{R2} \rangle, \langle 6, \mathcal{D}_{R3} \rangle, \langle 7, \mathcal{D}_{L2} \rangle\} \\ & \{\langle 0, \mathcal{D}_{R1} \rangle, \langle 1, \mathcal{D}_{R2} \rangle, \langle 2, \mathcal{D}_{R1} \rangle, \langle 3, \mathcal{D}_{R1} \rangle, \langle 4, \mathcal{D}_{L1} \rangle, \langle 5, \mathcal{D}_{R2} \rangle, \langle 6, \mathcal{D}_{R3} \rangle, \langle 7, \mathcal{D}_{L2} \rangle\} \\ & \{\langle 0, \mathcal{D}_{R1} \rangle, \langle 1, \mathcal{D}_{R1} \rangle, \langle 2, \mathcal{D}_{R3} \rangle, \langle 3, \mathcal{D}_{R1} \rangle, \langle 4, \mathcal{D}_{L1} \rangle, \langle 5, \mathcal{D}_{R2} \rangle, \langle 6, \mathcal{D}_{R3} \rangle, \langle 7, \mathcal{D}_{L2} \rangle\} \\ & \{\langle 0, \mathcal{D}_{R1} \rangle, \langle 1, \mathcal{D}_{R2} \rangle, \langle 2, \mathcal{D}_{R3} \rangle, \langle 3, \mathcal{D}_{R1} \rangle, \langle 4, \mathcal{D}_{L1} \rangle, \langle 5, \mathcal{D}_{R2} \rangle, \langle 6, \mathcal{D}_{R3} \rangle, \langle 7, \mathcal{D}_{L2} \rangle\}\ \} \end{aligned}$$

Observe that each of the above mappings associate each input of the truth function with a rule premise.

Having formed the topmost cut, the task now is to build the two cuts which will construct its left and right premise. For the first variant of the topmost cut given in Example 6.4.1 (with cut formula A_1), we would seek a cut that would build the bitmask $1XX$ from the premises $\mathcal{D}_{L1}, \mathcal{D}_{R1}, \mathcal{D}_{R2}, \mathcal{D}_{L2}$ (which cover the rows 4,5,6 and 7). The choice of cut formula for the topmost cut *freezes* the truth value of the first formula, and so the problem to be solved now can be compared to that of finding the topmost cut of an arity-two connective (i.e., finding the cut that

would build a bitmask with two 'don't care' elements ignoring the truth value of the element at index 0 of the bitmask). There are two variants since there are only two choices of cut formula from which to build the cut. The left and right pair of bitmasks for the two variants are: $\langle 10X, 11X\rangle$ and $\langle 1X0, 1X1\rangle$. Notice that applying `cut` to the components of either of these pairs of bitmasks would build the bitmask $1XX$. Now the sub-problems that need to be solved is building the bitmasks that are the components of one of the pairs, i.e., building a bitmask with exactly one truth value set to 'don't care'. We can solve this immediately by observing that it is the conclusion of a cut between two row inputs. For example, in the $10X$ case, the pair of left and right bitmasks would be $\langle 100, 101\rangle$. Now we can look up rows 100_2 and 101_2 (i.e., rows 4 and 5) in the base map and obtain a pair of premises. Retracting the steps taken, we can apply the cut rule to the pair of corresponding rule premises and build a right-hand side. The following procedure `split` formally specifies how we can split a 'root' bitmask and obtain all pair of bitmasks which, when cut together, rebuild that root bitmask. (In a sense, the procedure `split` is the reverse of the `cut` procedure).

Definition 6.4.4 (Splitting Bitmasks, `split`) *Given a length-n bitmask b which has k 'don't care' truth values at the set of positions P, i.e.,*

$$P = \{p \mid 0 \leq p < n \wedge b[p] = \text{'don't care'}\}$$

we define a function `split` *which builds the set of all pairs of bitmasks that have* $(k-1)$ *truth values set to 'don't care'. The function returns a 3-tuple consisting of: (1) the index on which the pair was split, (2) a modified bitmask* b_L *which has the truth value at that index equal to 0, and (3) a modified bitmask* b_R *which has the truth value at that index equal to 1.*

$$\begin{aligned}
&\texttt{split} :: \texttt{bitmask} \rightarrow \{\texttt{int}\} \rightarrow (\texttt{int} \times \texttt{bitmask} \times \texttt{bitmask}) \\
&\texttt{split}\ b\ P = \{\langle i, (\texttt{setelem}\ b\ i\ 0), (\texttt{setelem}\ b\ i\ 1)\rangle \mid i \in P\}
\end{aligned}$$

The auxiliary function `setelem` *sets the element at the specified position of the bitmask to the supplied truth value, and is defined as follows:*

$$\begin{aligned}
&\texttt{setelem} :: \texttt{bitmask} \rightarrow \texttt{int} \rightarrow \mathcal{T}^X \rightarrow \texttt{bitmask} \\
&\texttt{setelem}\ [\,]\ i\ t \quad\quad = [\,] \\
&\texttt{setelem}\ m{:}mask\ 0\ t = t{:}mask \\
&\texttt{setelem}\ m{:}mask\ i\ t\ = \texttt{setelem}\ mask\ (i-1)\ t
\end{aligned}$$

The `split` procedure can be used to build (all permutations of) trees of length-n bitmasks that are rooted at $X_1 \ldots X_n$, and have 'leaf' bitmasks with zero 'don't care' elements. A base map for an arity n connective would associate each leaf of this tree with a single rule premise. By working upwards from the leaves (applying the `cut` procedure) the right-hand side of the principal reduction rule can be constructed.

There are two problems with the above (outline of an) algorithm. Recall the set of base maps for the connective $\complement_9^3$ in Example 6.4.3. The rule premise $\mathcal{D}_{R1}$ (with bitmask $0XX$) covers the rows $0, 1, 2$ and 3. In the case where the bitmask $0XX$ is a node of the tree generated by repeatedly applying `split`, and the base map associates the rows $0, 1, 2, 3$ with the rule premise $\mathcal{D}_{R1}$, it would be incorrect to apply the `split` procedure to the bitmask $0XX$. The generated tree would have rows $0, 1, 2$ and 3 as leaves and each leaf would be associated with the same rule premise. A cut cannot be formed between two identical bitmasks since their `disagreecount` will be 0, so before applying `split` to a bitmask, we need to check whether the set of rows covered by that bitmask is covered completely by any single rule premise in the base map. In the running example, this check would ensure the bitmask $0XX$ is not split since the premise $\mathcal{D}_{R1}$ already covers all of the rows of `expand` $0XX$. We define the function `getPremiseForBitmask` below which performs this test.

Definition 6.4.5 (The procedure `getPremiseForBitmask`) *Given a bitmask b and a base map M, the function* `getPremiseForBitmask` *tests whether a derivation scheme in M covers all the rows covered by the bitmask b. The function returns the (possibly empty) set of derivation schemes which cover the rows covered by b. Note that the returned set will either contain a single derivation scheme, or be empty.*

$$\texttt{getPremiseForBitmask} :: \texttt{bitmask} \to \{\texttt{int} \times \texttt{dscheme}\} \to \{\texttt{dscheme}\}$$
$$\texttt{getPremiseForBitmask}\ b\ M = \{s \mid \exists ds.(\forall r \in \texttt{expand}\ b.(\langle r, s\rangle \in M \wedge s = ds))\}$$

Finally, we will address the problem of generating only right-linear rules. This is in fact fairly straightforward.

The application of the procedure `split` to an n-length bitmask builds a tree where: the root is the supplied bitmask, the left branch of the tree is the bitmask with the 'don't care' element at index i (for $0 \leq i < n$) set to 0, and the right branch of the tree is the bitmask with the 'don't care' element at index i set to 1. (In the corresponding derivation scheme, this structure would correspond to an instance of the cut rule with the cut formula A_{i+1}). The constructed tree will duplicate a

rule premise (and therefore not be right-linear) if the base map associates some common premise with the rows covered by both the left and right bitmasks. The following predicate tests for this case.

Definition 6.4.6 (A Predicate to Test for 'Good' Rules, `willDuplicate`) *A base map M for a connective $\complement_i^n$ associates each of the n rows of the truth table $\mathbb{C}_i^n$ with a rule premise of $(\complement_i^n L)$ and $(\complement_i^n R)$. Given a pair of bitmasks b_L and b_R that cover the set of rows R_L and R_R respectively, the predicate* `willDuplicate` *returns true if the base map associates some rule premise with both a row in R_L and a row in R_R. More formally,*

$$\begin{aligned}
&\texttt{willDuplicate} :: \texttt{bitmask} \to \texttt{bitmask} \to \{\texttt{int} \times \texttt{dscheme}\} \to \texttt{boolean} \\
&\texttt{willDuplicate}\ b_L\ b_R\ M = \forall l \in \texttt{expand}\ b_L. \\
&\qquad\qquad\qquad\qquad \forall r \in \texttt{expand}\ b_R. \\
&\qquad\qquad\qquad\qquad\quad \exists ds.(\langle l, ds\rangle \in M \wedge \langle r, ds\rangle \in M)
\end{aligned}$$

Now we can combine the above procedures and build the algorithm which constructs the 'good' right-hand sides of principal reduction rules. We do this in two steps: first we define a procedure `build` which builds 'good' right-hand sides using a single base map. We then define the procedure `solutions` which enumerates over the set of unique base maps and applies the procedure `build`.

Definition 6.4.7 (Right-hand Sides for a Single Base Map) *We first generalise the procedure* `cut` *(from Definition 6.1.9) to operate on* sets *of derivation schemes as follows:*

$$\begin{aligned}
&\texttt{cutSet} :: \{\texttt{dscheme}\} \to \{\texttt{dscheme}\} \to \texttt{formula} \to \{\texttt{dscheme}\} \\
&\texttt{cutSet}\ L\ R\ f = \{\texttt{cut}\ l\ r\ f \mid l \in L \wedge r \in R\}
\end{aligned}$$

Given a length-n bitmask b which has non-'don't care' truth values at indexes in F (c.f., a set of indexes for 'frozen' truth values of the bitmask), and a base map M mapping the rows of a truth table $\mathbb{C}_i^n$ to the premises of the associated inference rules $(\complement_i^n L)$ and $(\complement_i^n R)$, we define the function `build` *which constructs a set of 'good' right-hand sides of principal*

reduction rules as follows.

$$
\begin{aligned}
&\texttt{build} :: \texttt{bitmask} \to \{\texttt{int}\} \to \{\texttt{int} \times \texttt{dscheme}\} \to \{\texttt{dscheme}\} \\
&\texttt{build}\ b\ F\ M \mid (\mathit{CoversBitmask} \neq \emptyset) = \mathit{CoversBitmask} \\
&\qquad\qquad \mid \texttt{otherwise} = \bigcup_{\substack{\langle i,L,R\rangle \in \texttt{split}\ b\ F \\ \neg\texttt{willDuplicate}\ L\ R\ M}} \texttt{cutSet}\ \mathit{TreesForL}\ \mathit{TreesForR}\ A_{i+1} \\
&\qquad\qquad\quad \texttt{where}\ \mathit{CoversBitmask} = \texttt{getPremiseForBitmask}\ b\ M \\
&\qquad\qquad\qquad\quad \mathit{TreesForL} \quad = \texttt{build}\ L\ (F \cup \{i\})\ M \\
&\qquad\qquad\qquad\quad \mathit{TreesForR} \quad = \texttt{build}\ R\ (F \cup \{i\})\ M
\end{aligned}
$$

Definition 6.4.8 (Solutions) *Given a pair of inference rules $(\complement_i^n L)$ and $(\complement_i^n R)$ for the logical connective $\complement_i^n$, we enumerate the set of all right-hand sides of principal reduction rules by applying the* `build` *algorithm to each unique base map B associating each row of the truth table $\mathbb{C}_i^n$ to a single premise from the set of rule premises S.*

$$
\begin{aligned}
&\texttt{solutions} :: \{\texttt{dscheme}\} \to \texttt{int} \to \{\texttt{dscheme}\} \\
&\texttt{solutions}\ S\ n = \bigcup_{B \in \texttt{baseMaps}\ S\ n} \texttt{build}\ X_1 \ldots X_n\ \emptyset\ B
\end{aligned}
$$

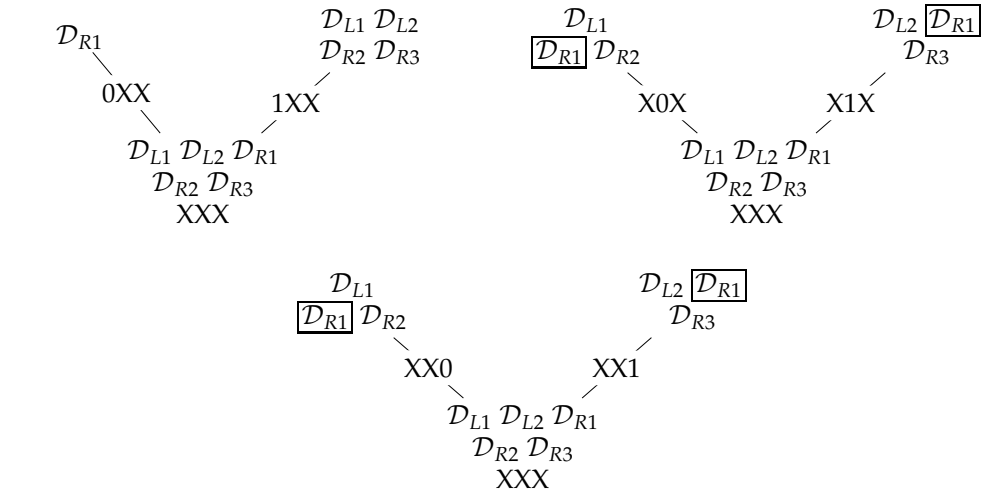
We will illustrate a use of the above algorithm in the following example.

Example 6.4.9 (Constructing the `solutions` of $\complement_9^3$) *We apply the* `solutions` *procedure to the premises of the inference rules $(\complement_9^3 L)$ and $(\complement_9^3 R)$ for the arity three connective $\complement_9^3$. This example is accompanied by Figures 6.3, 6.4, 6.5 and 6.6, which illustrate how a base-map is partitioned when the* `split`*ting steps of the algorithm are applied. Recall from Example 6.4.3 that the set S of rule premises for the connective $\complement_9^3$ (using the abbreviated form of writing derivation schemes) is,*

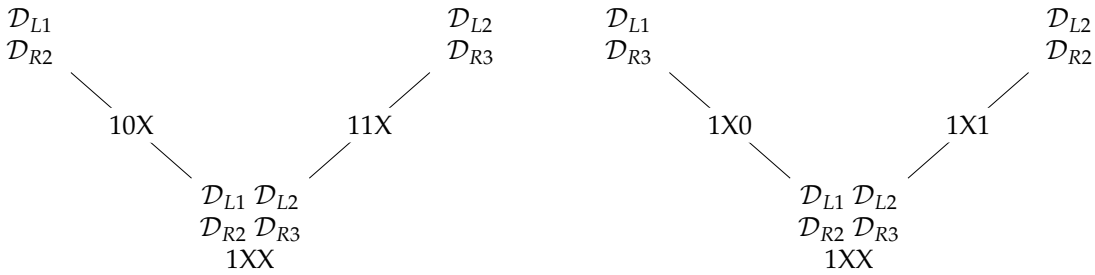
$$S = \{\mathcal{D}_{L1}, \mathcal{D}_{L2}, \mathcal{D}_{R1}, \mathcal{D}_{R2}, \mathcal{D}_{R3}\}$$

We also computed the set of all base maps for S. Recall that these were:

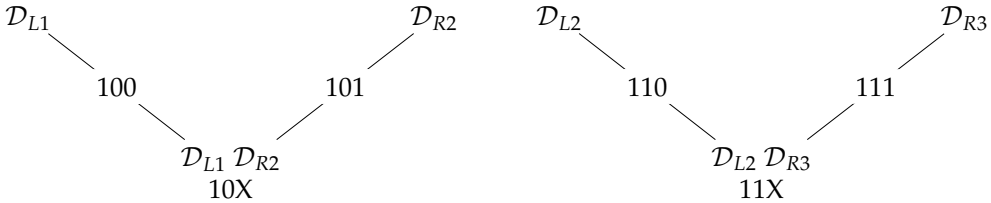
$$
\begin{aligned}
&\texttt{baseMaps}\ S\ 3 \\
&= \{\{M_1\}, \{M_2\}, \{M_3\}, \{M_4\}\} \\
&= \{\ \{\langle 0, \mathcal{D}_{R1}\rangle, \langle 1, \mathcal{D}_{R1}\rangle, \langle 2, \mathcal{D}_{R1}\rangle, \langle 3, \mathcal{D}_{R1}\rangle, \langle 4, \mathcal{D}_{L1}\rangle, \langle 5, \mathcal{D}_{R2}\rangle, \langle 6, \mathcal{D}_{R3}\rangle, \langle 7, \mathcal{D}_{L2}\rangle\} \\
&\quad\ \{\langle 0, \mathcal{D}_{R1}\rangle, \langle 1, \mathcal{D}_{R2}\rangle, \langle 2, \mathcal{D}_{R1}\rangle, \langle 3, \mathcal{D}_{R1}\rangle, \langle 4, \mathcal{D}_{L1}\rangle, \langle 5, \mathcal{D}_{R2}\rangle, \langle 6, \mathcal{D}_{R3}\rangle, \langle 7, \mathcal{D}_{L2}\rangle\} \\
&\quad\ \{\langle 0, \mathcal{D}_{R1}\rangle, \langle 1, \mathcal{D}_{R1}\rangle, \langle 2, \mathcal{D}_{R3}\rangle, \langle 3, \mathcal{D}_{R1}\rangle, \langle 4, \mathcal{D}_{L1}\rangle, \langle 5, \mathcal{D}_{R2}\rangle, \langle 6, \mathcal{D}_{R3}\rangle, \langle 7, \mathcal{D}_{L2}\rangle\} \\
&\quad\ \{\langle 0, \mathcal{D}_{R1}\rangle, \langle 1, \mathcal{D}_{R2}\rangle, \langle 2, \mathcal{D}_{R3}\rangle, \langle 3, \mathcal{D}_{R1}\rangle, \langle 4, \mathcal{D}_{L1}\rangle, \langle 5, \mathcal{D}_{R2}\rangle, \langle 6, \mathcal{D}_{R3}\rangle, \langle 7, \mathcal{D}_{L2}\rangle\}\ \}
\end{aligned}
$$

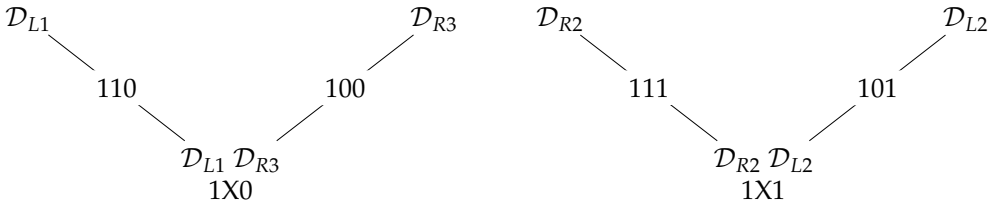


(a) Top-Level Splitting



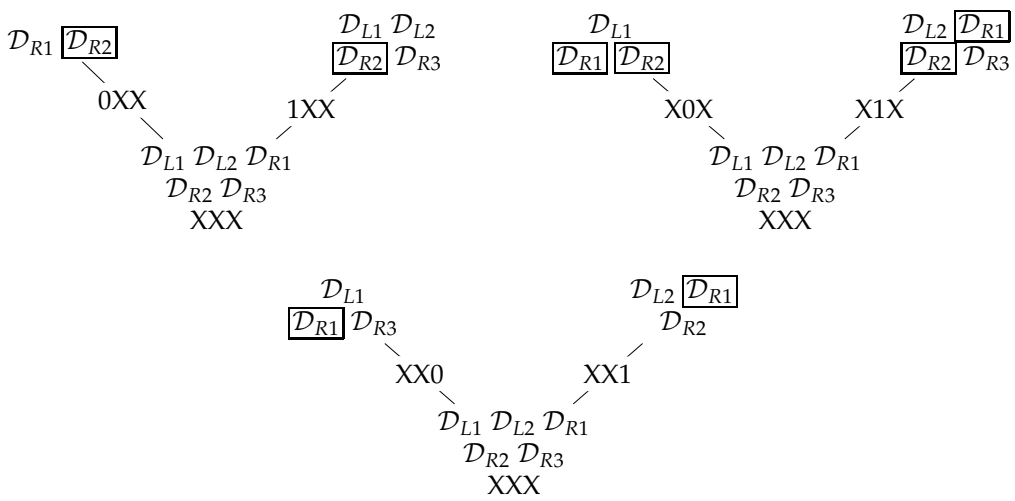
(b) Second Level Splittings





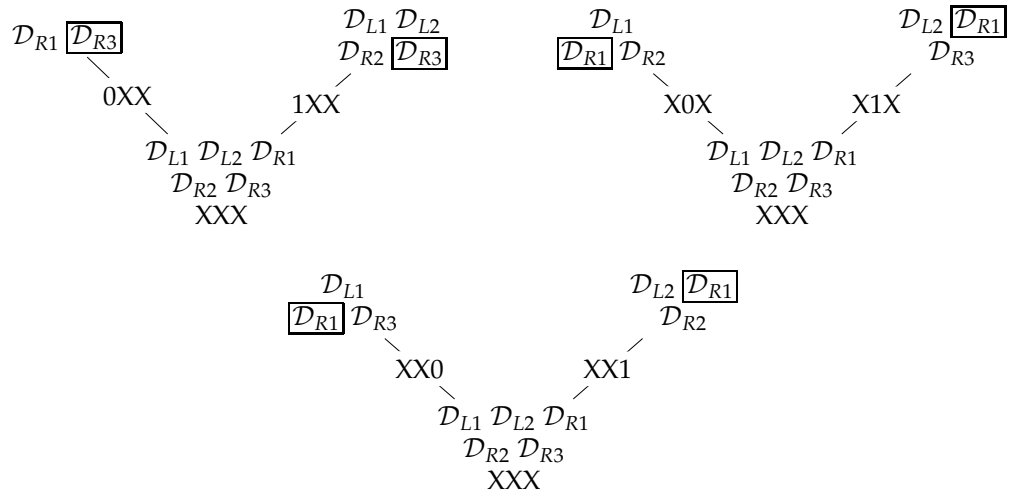
(c) Third Level Splittings

Figure 6.3: 'Splittings' for Building Right-Hand Sides of $(\mathbb{C}_9^3)$ using Base-Map M_1



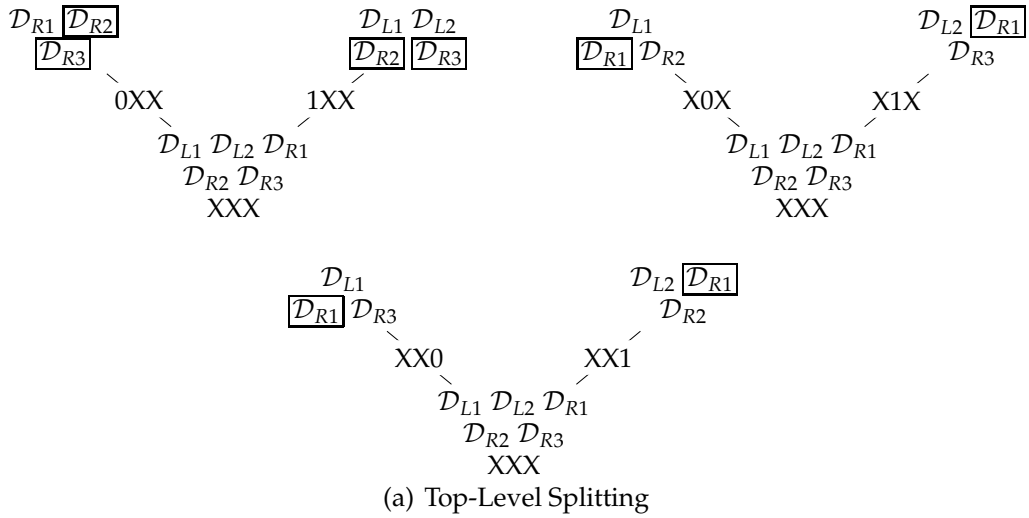
(a) Top-Level Splitting

Figure 6.4: 'Splittings' for Building Right-Hand Sides of $(\mathbb{C}_9^3)$ using Base-Map M_2



(a) Top-Level Splitting

Figure 6.5: 'Splittings' for Building Right-Hand Sides of $(\mathbb{C}_9^3)$ using Base-Map M_3



(a) Top-Level Splitting

Figure 6.6: 'Splittings' for Building Right-Hand Sides of $(\mathbb{C}_9^3)$ using Base-Map M_4

Now we compute the set of all 'good' right-hand side of principal reduction rules as follows,

$$\begin{aligned}\texttt{solutions}\ S\ 3 = {} & (\texttt{build}\ XXX\ \emptyset\ M_1) \cup \\ & (\texttt{build}\ XXX\ \emptyset\ M_2) \cup \\ & (\texttt{build}\ XXX\ \emptyset\ M_3) \cup \\ & (\texttt{build}\ XXX\ \emptyset\ M_4)\end{aligned}$$

Where the four applications of build are computed as follows,

1. *For the call* $\texttt{build}\ XXX\ \emptyset\ M_1$*, no single premise in the base map* M_1 *covers all the rows of* XXX*, i.e.,*

   $$\mathit{CoversBitmask} = \texttt{getPremiseForBitmask}\ XXX\ M_1 = \emptyset$$

   We apply `split` *to* XXX *and obtain a bitmask covering fewer rows. (Notice that the set of 'frozen' indexes of the bitmask is empty, and all indexes of the bitmask are set to 'don't care').*

   $$\texttt{split}\ XXX\ \emptyset = \{\langle 0, 0XX, 1XX\rangle, \langle 1, X0X, X1X\rangle, \langle 2, XX0, XX1\rangle\}$$

   To ensure the generated right-hand side is 'good', we must prune the above set to ensure no rule premise is copied as a result of the `split`*. Observe that there is only one 'good' choice.*

   $$\begin{aligned}\texttt{willDuplicate}\ 0XX\ 1XX\ M_1 &= \mathit{false} \\ \texttt{willDuplicate}\ X0X\ X1X\ M_1 &= \mathit{true} \\ \texttt{willDuplicate}\ XX0\ XX1\ M_1 &= \mathit{true}\end{aligned}$$

   Calculating $\texttt{expand}\ X0X = \{0, 1, 4, 5\}$ *and* $\texttt{expand}\ X1X = \{2, 3, 6, 7\}$*, notice that* M_1 *maps both rows* 0 *and* 2 *to the rule premise* $\mathcal{D}_{R1}$*, so splitting on index* 1 *would cause the rule premise* $\mathcal{D}_{R1}$ *to be copied. Also,* $\texttt{expand}\ XX0 = \{0, 2, 4, 6\}$*,* $\texttt{expand}\ XX1 = \{1, 3, 5, 7\}$ *and* M_1 *maps both rows 0 and 1 to the rule premise* $\mathcal{D}_{R1}$*, causing it to be copied in the solution. Therefore, using the base map* M_1*, there is only one 'good' choice for the topmost cut of the right-hand side: it must be built with the cut formula* A_1*. This is illustrated in Figure 6.3(a); the square boxes show which rule premise(s) would be copied if the derivation scheme was built accordingly.*

   Now we compute the left and right premises which correspond to the bitmasks $0XX$

and $1XX$ *respectively. Computing the left premise is easy. Observe that,*

$$\begin{aligned} \mathit{TreesForL} &= \texttt{build}\ 0XX\ \{0\}\ M_1 \\ &= \mathit{CoversBitmask} \\ &= \texttt{getPremiseForBitmask}\ 0XX\ M_1 \\ &= \{\mathcal{D}_{R1}\} \end{aligned}$$

In other words, the set of rule premises $\{\mathcal{D}_{R1}\}$ *already covers all of the rows covered by the bitmask* $0XX$*. This means that there is no need to apply* `split` *to try and obtain simpler bitmasks.*

Computing the right premise for the cut with conclusion $1XX$*, we have:*

$$\mathit{TreesForR} = \texttt{build}\ 1XX\ \{0\}\ M_1 \tag{6.7}$$

This time $\texttt{getPremiseForBitmask}\ 1XX\ M_1 = \emptyset$*, i.e., there is no single rule premise in the base map that covers all the rows covered by* $1XX$*. We apply* `split`*, but notice this time that the truth value at index* 0 *of the bitmask has been frozen; recall that this is because the the cut formula of the topmost cut was chosen to be* A_1*.*

$$\texttt{split}\ 1XX\ \{0\} = \{\langle 1, 10X, 11X\rangle, \langle 2, 1X0, 1X1\rangle\}$$

Both of the above 'splittings' will produce 'good' right-hand sides, i.e.,

$$\begin{aligned} \texttt{willDuplicate}\ 10X\ 11X\ M_1 &= \mathit{false} \\ \texttt{willDuplicate}\ 1X0\ 1X1\ M_1 &= \mathit{false} \end{aligned}$$

This is shown in Figure 6.3(b); notice that no rule premise is duplicated. We recursively call the `build` *procedure on each bitmask while freezing the relevant index of the bitmask. There are two cases to consider: splitting on index 1 and splitting on index 2.*

First splitting on index 1 *of the bitmask (and subsequently freezing the truth value at index 1), we have two procedure calls:*

(a) Call $\texttt{build}\ 10X\ \{0,1\}\ M_1$*. Here* $\mathit{CoversBitmask} = \emptyset$*, indicating that the rows covered by* $10X$ *are covered by more than one rule premise, so we must again apply* `split` *to the bitmask. There is only one way to split this bitmask, since there is only one 'don't care' state and therefore only one choice of cut formula:*

$$\texttt{split}\ 10X\ \{0,1\} = \{\langle 2, 100, 101\rangle\}$$

Notice also that `willDuplicate` *will return false, since the base-map* M_1

maps rows 100_2 *and* 101_2 *to the different rule premises (see top-left illustration of Figure 6.3(c)). A final pair of recursive calls to* `build` *returns two sets of derivations schemes which cover the rows covered by* 100_2 *and* 101_2*. i.e.,*

$$\begin{aligned}\texttt{build}\ 100\ \{0,1,2\}M_1 &= \textit{CoversBitmask}\\ &= \texttt{getPremiseForBitmask}\ 100\ M_1\\ &= \{\mathcal{D}_{L1}\}\end{aligned}$$

$$\begin{aligned}\texttt{build}\ 101\ \{0,1,2\}M_1 &= \textit{CoversBitmask}\\ &= \texttt{getPremiseForBitmask}\ 101\ M_1\\ &= \{\mathcal{D}_{R2}\}\end{aligned}$$

So the result of the initial call to `build` *for the bitmask* $10X$ *is computed as:*

$$\begin{aligned}&\texttt{build}\ 10X\ \{0,1\}\ M_1\\ =\ &\texttt{cutSet}\ \{\mathcal{D}_{L1}\}\ \{\mathcal{D}_{R2}\}\ A_3\\ =\ &\{\langle(A_1 \vdash A_2), [\mathcal{D}_{L1}, \mathcal{D}_{R2}]\rangle\}\end{aligned}$$

Where the cut formula is chosen to be A_3*, since this corresponds to the formula whose truth value is disagreed upon by the two bitmasks.*

(b) Call `build` $11X\ \{0,1\}\ M_1$*. Again* $\textit{CoversBitmask} = \emptyset$ *indicating there is no premise in the bitmask that covers all of the rows in the set* `expand` $11X$*. We apply* `split` *to obtain bitmasks which cover fewer rows, i.e.,*

$$\texttt{split}\ 11X\ \{0,1\} = \{\langle 2, 110, 111\rangle\}$$

The splitting will build a 'good' right-hand side, since applying `willDuplicate` *to each bitmask returns false (see the right-hand illustration of Figure 6.3(b)). Applying* `build` *to these bitmasks one last time, we have (as shown in top-right illustration of Figure 6.3(c)):*

$$\begin{aligned}\texttt{build}\ 110\ \{0,1,2\}\ M_1 &= \textit{CoversBitmask}\\ &= \texttt{getPremiseForBitmask}\ 110\ M_1\\ &= \{\mathcal{D}_{R3}\}\end{aligned}$$

$$\begin{aligned}\texttt{build}\ 111\ \{0,1,2\}\ M_1 &= \textit{CoversBitmask}\\ &= \texttt{getPremiseForBitmask}\ 111\ M_1\\ &= \{\mathcal{D}_{L2}\}\end{aligned}$$

So the result of the initial call to `build` *is computed as:*

$$\begin{aligned} & \texttt{build}\ 11X\ \{0,1\}\ M_1 \\ = {} & \texttt{cutSet}\ \{\mathcal{D}_{R3}\}\ \{\mathcal{D}_{L2}\}\ A_3 \\ = {} & \{\langle (A_1, A_2 \vdash), [\mathcal{D}_{R3}, \mathcal{D}_{L2}]\rangle\} \end{aligned}$$

We also have two procedure calls to deal with for the second case of splitting the bitmask $1XX$ *on the index* 2*.*

(a) Call `build` $1X0\ \{0,2\}\ M_1$*. Following the same steps as described in parts (a) and (b) above, we have,*

$$\begin{aligned} & \texttt{build}\ 1X0\ \{0,2\}\ M_1 \\ = {} & \texttt{cutSet}\ \{\mathcal{D}_{L1}\}\ \{\mathcal{D}_{R3}\}\ A_2 \\ = {} & \{\langle (A_1 \vdash A_3), [\mathcal{D}_{L1}, \mathcal{D}_{R3}]\rangle\} \end{aligned}$$

(b) Call `build` $1X1\ \{0,2\}\ M_1$*. Similarly, we have,*

$$\begin{aligned} & \texttt{build}\ 1X1\ \{0,2\}\ M_1 \\ = {} & \texttt{cutSet}\ \{\mathcal{D}_{R2}\}\ \{\mathcal{D}_{L2}\}\ A_2 \\ = {} & \{\langle (A_1, A_3 \vdash), [\mathcal{D}_{R2}, \mathcal{D}_{L2}]\rangle\} \end{aligned}$$

The above are shown in the bottom two illustrations of Figure 6.3(c).

Tracing back to Equation (6.7)*, we combine the above recursive calls to* `build`*, as follows:*

$$\begin{aligned} & \textit{TreesForR} \\ = {} & \texttt{build}\ 1XX\ \{0\}\ M_1 \\ = {} & \texttt{cutSet}\ \{\langle (A_1 \vdash A_2), [\mathcal{D}_{L1}, \mathcal{D}_{R2}]\rangle\}\ \{\langle (A_1, A_2 \vdash), [\mathcal{D}_{R3}, \mathcal{D}_{L2}]\rangle\}\ A_2 \cup \\ & \texttt{cutSet}\ \{\langle (A_1 \vdash A_3), [\mathcal{D}_{L1}, \mathcal{D}_{R3}]\rangle\}\ \{\langle (A_1, A_3 \vdash), [\mathcal{D}_{R2}, \mathcal{D}_{L2}]\rangle\}\ A_3 \\ \\ = {} & \{\langle (A_1 \vdash), [\langle (A_1 \vdash A_2), [\mathcal{D}_{L1}, \mathcal{D}_{R2}]\rangle, \langle (A_1, A_2 \vdash), [\mathcal{D}_{R3}, \mathcal{D}_{L2}]\rangle]\rangle\} \cup \\ & \{\langle (A_1 \vdash), [\langle (A_1 \vdash A_3), [\mathcal{D}_{L1}, \mathcal{D}_{R3}]\rangle, \langle (A_1, A_3 \vdash), [\mathcal{D}_{R2}, \mathcal{D}_{L2}]\rangle]\rangle\} \\ \\ = {} & \{\langle (A_1 \vdash), [\langle (A_1 \vdash A_2), [\mathcal{D}_{L1}, \mathcal{D}_{R2}]\rangle, \langle (A_1, A_2 \vdash), [\mathcal{D}_{R3}, \mathcal{D}_{L2}]\rangle]\rangle \\ & \quad \langle (A_1 \vdash), [\langle (A_1 \vdash A_3), [\mathcal{D}_{L1}, \mathcal{D}_{R3}]\rangle, \langle (A_1, A_3 \vdash), [\mathcal{D}_{R2}, \mathcal{D}_{L2}]\rangle]\rangle\} \\ = {} & \{\mathcal{D}_{TM_{R1}}, \mathcal{D}_{TM_{R2}}\} \end{aligned}$$

The above set gives two *derivation (scheme) trees for constructing the right premise of the topmost cut. Now that we have successfully computed the left and right sets of premises which, when cut together, produce the topmost cut split in* A_1*, we can*

combine the results and build right-hand sides as follows:

$$
\begin{aligned}
&= \texttt{cutSet}\ \{\mathcal{D}_{R1}\}\ \{\mathcal{D}_{TM_{R1}}, \mathcal{D}_{TM_{R2}}\}\ A_1 \\
&= \{\langle (\vdash), [\mathcal{D}_{R1}, \mathcal{D}_{TM_{R1}}]\rangle, \langle (\vdash), [\mathcal{D}_{R1}, \mathcal{D}_{TM_{R2}}]\rangle\}
\end{aligned}
$$

2. *Call* `build` $XXX\ \emptyset\ M_2$*: The base map M_2 does not produce any 'good' solutions. See Figure 6.4(a).*

3. *Call* `build` $XXX\ \emptyset\ M_3$*: The base map M_3 does not produce any 'good' solutions. See Figure 6.5(a).*

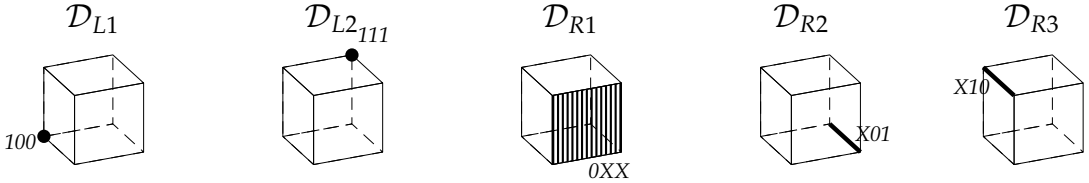
4. *Call* `build` $XXX\ \emptyset\ M_4$*: The base map M_4 does not produce any 'good' solutions. See Figure 6.6(a).*

So, in sequent calculus form, the 'good' right-hand sides of the principal reduction rules for the logical connective are:

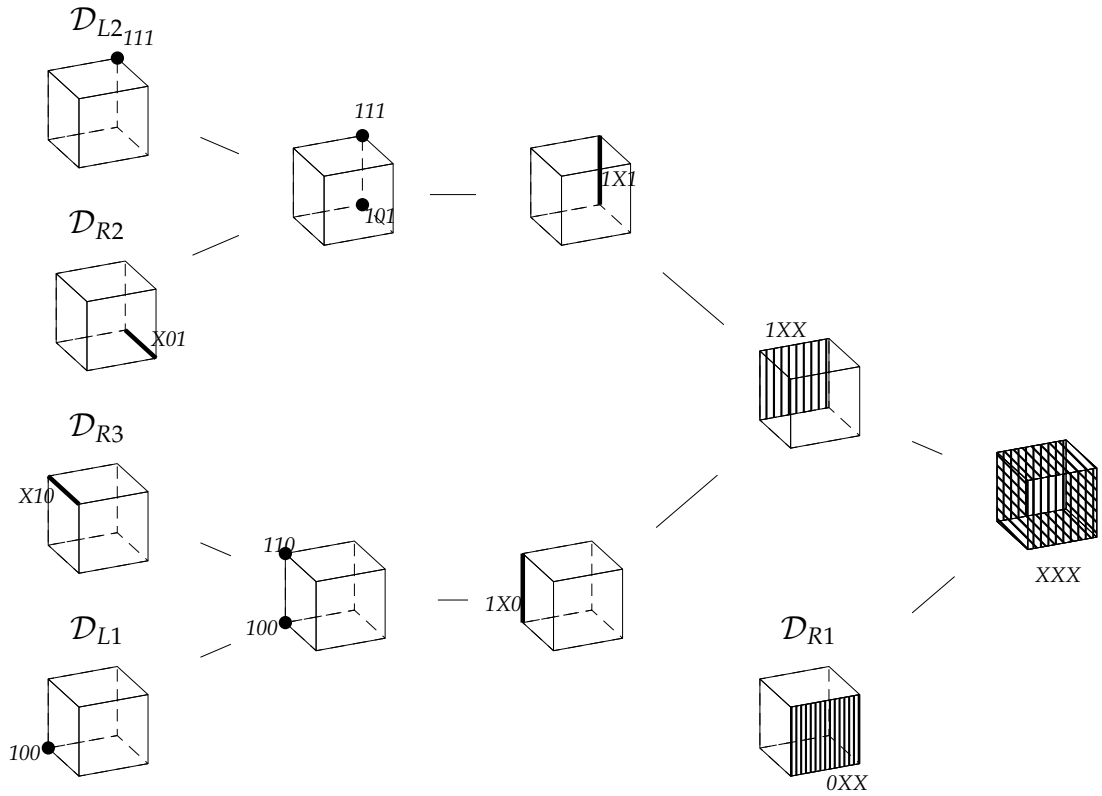
$$
\dfrac{\dfrac{\boxed{D_{R1}}}{\Gamma \vdash \Delta, A_1} \qquad \dfrac{\dfrac{\boxed{D_{L1}}}{A_1, \Gamma \vdash \Delta, A_2, A_3} \quad \dfrac{\boxed{D_{R2}}}{A_3, \Gamma \vdash \Delta, A_2}\,(Cut)}{A_1, \Gamma \vdash \Delta, A_2} \qquad \dfrac{\dfrac{\boxed{D_{R3}}}{A_2, \Gamma \vdash \Delta, A_3} \quad \dfrac{\boxed{D_{L2}}}{A_1, A_2, A_3, \Gamma \vdash \Delta}\,(Cut)}{A_1, A_2, \Gamma \vdash \Delta}\,(Cut)}{A_1, \Gamma \vdash \Delta}\,(Cut)}{\Gamma \vdash \Delta}\,(Cut)
$$

$$
\dfrac{\dfrac{\boxed{D_{R1}}}{\Gamma \vdash \Delta, A_1} \qquad \dfrac{\dfrac{\boxed{D_{L1}}}{A_1, \Gamma \vdash \Delta, A_2, A_3} \quad \dfrac{\boxed{D_{R3}}}{A_2, \Gamma \vdash \Gamma, A_3}\,(Cut)}{A_1, \Gamma \vdash \Delta, A_3} \qquad \dfrac{\dfrac{\boxed{D_{R2}}}{A_3, \Gamma \vdash \Delta, A_2} \quad \dfrac{\boxed{D_{L2}}}{A_1, A_2, A_3, \Gamma \vdash \Delta}\,(Cut)}{A_1, A_3, \Gamma \vdash \Delta}\,(Cut)}{A_1, \Gamma \vdash \Delta}\,(Cut)}{\Gamma \vdash \Delta}\,(Cut)
$$

Example 6.4.10 (Merging of $\mathbb{C}_9^3$ Hypercubes) *As a final (and interesting example), we show how to generate the right-hand side of a principal reduction rule for $\mathbb{C}_9^3$ (from the previous example) by 'merging' together hypercubes. We begin with by representing each premise of the rules $(\mathbb{C}_9^3 L)$ and $(\mathbb{C}_9^3 R)$ (see Example 6.2.1) as a hypercube (mapped onto the skeleton 3-cube we gave in Figure 6.2).*



Now we proceed by taking pairs of hypercubes and 'merging' them according to the specification we described at the end of Section 6.3.



We conclude this section with the definition of the term calculus $\mathcal{X}^{\mathbb{C}_9^3}$-calculus, since after all our main goal was to show that we could automate the construction of Curry-Howard pairs of logical and (corresponding) computational calculi. The full definition of the calculus is given in Figure 6.7.

6.5 Chapter Summary

In this chapter we gave a formal specification for Call's algorithm, also describing our own intuitions behind his mechanical process (which builds a pair of sequent calculus style inference rules from the truth table of a logical connective). We were able to construct a reverse process which associated the rule premises of an inference rule for a connective to a *set* of rows from the truth table for that connective. This gave us important insight into the operation of the (Cut) rule, and also allowed us to make a relation between the cut and geometry.

Using results from the reverse algorithm, we described an intelligent and systematic algorithm which built the principal reduction rules (or 'cut elimination rules') for that logical connective. We focused on building only *good* principal reduction rules, i.e., those which did not copy any rule premise.

Definition 6.4.11 ($\mathcal{X}^{\mathsf{C}^3_9}$-Syntax) *The circuits of the $\mathcal{X}^{\mathsf{C}^3_9}$-calculus are defined by the following grammar, where x, y range over the infinite set of* sockets*, and α, β over* plugs*.*

$$P, Q ::= \underset{\text{capsule}}{\langle x{\cdot}\alpha\rangle} \mid \underset{\mathsf{C}^3_9 \text{ input circuit}}{y{\cdot}[\widehat{x}M\widehat{\alpha}\widehat{\beta}, \widehat{u}\widehat{v}\widehat{w}N]} \mid \underset{\mathsf{C}^3_9 \text{ output circuit}}{[P\widehat{\sigma}, \widehat{t}Q\widehat{\pi}, \widehat{s}R\widehat{\delta}]{\cdot}\gamma} \mid \underset{\text{cut}}{P\widehat{\alpha} \dagger \widehat{x}Q}$$

Definition 6.4.12 (Typing Rules for $\mathcal{X}^{\mathsf{C}^3_9}$) *The axiom and cut are typed as usual (Definition 5.1.3). The C^3_9 input and output circuits are typed as follows.*

$$\frac{M :\cdot\ x{:}A_1, \Gamma \vdash \Delta, \alpha{:}A_2, \beta{:}A_3 \qquad N :\cdot\ u{:}A_1, v{:}A_2, w{:}A_3, \Gamma \vdash \Delta}{y{\cdot}[\widehat{x}M\widehat{\alpha}\widehat{\beta}, \widehat{u}\widehat{v}\widehat{w}N] :\cdot\ \mathsf{C}^3_9(A_1, A_2, A_3), \Gamma \vdash \Delta}\ (\mathsf{C}^3_9 L)$$

$$\frac{P :\cdot\ \Gamma \vdash \Delta, \sigma{:}A_1 \qquad Q :\cdot\ t{:}A_2, \Gamma \vdash \Delta, \pi{:}A_3 \qquad R :\cdot\ s{:}A_3, \Gamma \vdash \Delta, \delta{:}A_2}{[P\widehat{\sigma}, \widehat{t}Q\widehat{\pi}, \widehat{s}R\widehat{\delta}]{\cdot}\gamma :\cdot\ \Gamma \vdash \Delta, \mathsf{C}^3_9(A_1, A_2, A_3)}\ (\mathsf{C}^3_9 R)$$

Definition 6.4.13 ($\mathcal{X}^{\mathsf{C}^3_9}$ Reduction Rules) *We extend the set of* basic reduction rules*, $\mathcal{R}$, (Definition 5.2.4), with the following reduction rules.*

***Left Propagation Rules** : $(\mathsf{C}^3_9 O\text{-}outs\not\leftarrow)$, $(\mathsf{C}^3_9 O\text{-}ins\not\leftarrow)$ and $(\mathsf{C}^3_9 I\not\leftarrow)$*

$$\begin{array}{l}
([P\widehat{\sigma}, \widehat{t}Q\widehat{\pi}, \widehat{s}R\widehat{\delta}]{\cdot}\gamma)\widehat{\gamma} \not\leftarrow \widehat{y}S \ \rightarrow\ ([(P\widehat{\gamma} \not\leftarrow \widehat{y}S)\widehat{\sigma}, \widehat{t}(Q\widehat{\gamma} \not\leftarrow \widehat{y}S)\widehat{\pi}, \widehat{s}(R\widehat{\gamma} \not\leftarrow \widehat{y}S)\widehat{\delta}]{\cdot}\beta)\widehat{\beta} \dagger \widehat{y}S \\
([P\widehat{\sigma}, \widehat{t}Q\widehat{\pi}, \widehat{s}R\widehat{\delta}]{\cdot}\alpha)\widehat{\gamma} \not\leftarrow \widehat{y}S \ \rightarrow\ [(P\widehat{\gamma} \not\leftarrow \widehat{y}S)\widehat{\sigma}, \widehat{t}(Q\widehat{\gamma} \not\leftarrow \widehat{y}S)\widehat{\pi}, \widehat{s}(R\widehat{\gamma} \not\leftarrow \widehat{y}S)\widehat{\delta}]{\cdot}\gamma \ \leftarrow \gamma \neq \alpha \\
(z{\cdot}[\widehat{x}M\widehat{\alpha}\widehat{\beta}, \widehat{u}\widehat{v}\widehat{w}N])\widehat{\gamma} \not\leftarrow \widehat{y}S \ \rightarrow\ z{\cdot}[\widehat{x}(M\widehat{\gamma} \not\leftarrow \widehat{y}S)\widehat{\alpha}\widehat{\beta}, \widehat{u}\widehat{v}\widehat{w}(N\widehat{\gamma} \not\leftarrow \widehat{y}S)]
\end{array}$$

***Right Propagation Rules** : $(\not\rightarrow \mathsf{C}^3_9 I\text{-}outs)$, $(\not\rightarrow \mathsf{C}^3_9 I\text{-}ins)$ and $(\not\rightarrow \mathsf{C}^3_9 O)$*

$$\begin{array}{l}
S\widehat{\gamma} \not\rightarrow \widehat{y}(y{\cdot}[\widehat{x}M\widehat{\alpha}\widehat{\beta}, \widehat{u}\widehat{v}\widehat{w}N]) \ \rightarrow\ S\widehat{\gamma} \dagger \widehat{k}(k{\cdot}[\widehat{x}(S\widehat{\gamma} \not\rightarrow \widehat{y}M)\widehat{\alpha}\widehat{\beta}, \widehat{u}\widehat{v}\widehat{w}(S\widehat{\gamma} \not\rightarrow \widehat{y}N)]) \\
S\widehat{\gamma} \not\rightarrow \widehat{y}(z{\cdot}[\widehat{x}M\widehat{\alpha}\widehat{\beta}, \widehat{u}\widehat{v}\widehat{w}N]) \ \rightarrow\ z{\cdot}[\widehat{x}(S\widehat{\gamma} \not\rightarrow \widehat{y}M)\widehat{\alpha}\widehat{\beta}, \widehat{u}\widehat{v}\widehat{w}(S\widehat{\gamma} \not\rightarrow \widehat{y}N)] \quad \leftarrow y \neq z \\
S\widehat{\gamma} \not\rightarrow \widehat{y}([P\widehat{\sigma}, \widehat{t}Q\widehat{\pi}, \widehat{s}R\widehat{\delta}]{\cdot}\gamma) \ \rightarrow\ [(S\widehat{\gamma} \not\rightarrow \widehat{y}P)\widehat{\sigma}, \widehat{t}(S\widehat{\gamma} \not\rightarrow \widehat{y}Q)\widehat{\pi}, \widehat{s}(S\widehat{\gamma} \not\rightarrow \widehat{y}R)\widehat{\delta}]{\cdot}\gamma
\end{array}$$

***Renaming Rules** : $(\mathsf{C}^3_9 I\text{-}rn)$ and $(\mathsf{C}^3_9 O\text{-}rn)$:*

$$\begin{array}{l}
\langle z{\cdot}\gamma\rangle\widehat{\gamma} \dagger \widehat{y}(y{\cdot}[\widehat{x}M\widehat{\alpha}\widehat{\beta}, \widehat{u}\widehat{v}\widehat{w}N]) \ \rightarrow\ z{\cdot}[\widehat{x}M\widehat{\alpha}\widehat{\beta}, \widehat{u}\widehat{v}\widehat{w}N] \ \leftarrow y \text{ introduced} \\
([P\widehat{\sigma}, \widehat{t}Q\widehat{\pi}, \widehat{s}R\widehat{\delta}]{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{y}\langle y{\cdot}\alpha\rangle \ \rightarrow\ [P\widehat{\sigma}, \widehat{t}Q\widehat{\pi}, \widehat{s}R\widehat{\delta}]{\cdot}\alpha \ \leftarrow \gamma \text{ introduced}
\end{array}$$

***Principal Reduction Rules** : (C^3_{91}) and (C^3_{92}), where γ, y are introduced,*

$$\begin{array}{l}
([P\widehat{\sigma}, \widehat{t}Q\widehat{\pi}, \widehat{s}R\widehat{\delta}]{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{y}(y{\cdot}[\widehat{x}M\widehat{\alpha}\widehat{\beta}, \widehat{u}\widehat{v}\widehat{w}N]) \rightarrow \\
\qquad P\widehat{\sigma} \dagger \widehat{x}(\langle x{\cdot}\epsilon\rangle\widehat{\epsilon} \dagger \widehat{u}(((M\widehat{\alpha} \dagger \widehat{t}Q)\widehat{\beta} \dagger \widehat{c}\langle c{\cdot}\pi\rangle)\widehat{\pi} \dagger \widehat{w}(\langle w{\cdot}\mu\rangle\widehat{\mu} \dagger \widehat{s}(R\widehat{\delta} \dagger \widehat{v}N))))
\end{array}$$

$$\begin{array}{l}
([P\widehat{\sigma}, \widehat{t}Q\widehat{\pi}, \widehat{s}R\widehat{\delta}]{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{y}(y{\cdot}[\widehat{x}M\widehat{\alpha}\widehat{\beta}, \widehat{u}\widehat{v}\widehat{w}N]) \rightarrow \\
\qquad P\widehat{\sigma} \dagger \widehat{x}(\langle x{\cdot}\epsilon\rangle\widehat{\epsilon} \dagger \widehat{u}(((M\widehat{\beta} \dagger \widehat{s}R)\widehat{\delta} \dagger \widehat{c}\langle c{\cdot}\alpha\rangle)\widehat{\alpha} \dagger \widehat{t}(\langle t{\cdot}\mu\rangle\widehat{\mu} \dagger \widehat{v}(Q\widehat{\pi} \dagger \widehat{w}N))))
\end{array}$$

Figure 6.7: The $\mathcal{X}^{\mathsf{C}^3_9}$-Calculus

Chapter 7

Conclusion

The work we have presented in thesis combines three distinct fields of computing: proof theory, computability theory and term rewriting. We related these three fields in our study of Curry-Howard correspondences.

We were interested mainly in studying the computational content of Classical Logics. Recently, a computational term calculus called $\mathcal{X}$ was introduced by van Bakel, Lengrand and Lescanne and shown to hold a close correspondence with a variant of (Kleene's refinement of) Gentzen's Sequent Calculus for Classical Logic.

We began our investigations with a review of the works that were related to the $\mathcal{X}$-calculus. On the proof theory side, we found out that the most natural presentations of classical logic were formulated in a (symmetric) sequent calculus. The most important property of the sequent calculus is unarguably its cut-elimination, which has a number of uses. For example, various authors have shown the preservation of the property could be used to guarantee extensions of logics are conservative. Next we looked at common ways in which classical logics were extended. A typical approach taken is to extend the logic with any number of primitive classical logical connectives (such connectives are semantically defined by two-valued truth functions). We found some works that presented mechanical methods for building sequent calculus style inference rules for logical connectives directly from truth functions. Two of these works (by Ciabattoni and Leitsch, and Baaz *et al.*) also presented algorithms to mechanically build local cut-elimination procedures and therefore ensured the extension was conservative. Both works noticed that the main difficulty was in defining the 'principal reduction rule'. The solutions proposed for building this rule were similar and were based on techniques which searched for the rule using a brute-force ap-

proach. We also observed that each algorithm built only one permutation of the main cut-elimination rule for the connective in question, even though several permutations could have existed. We remarked that in the specific context of proof theory, there was no obvious reason to consider multiple permutations.

On the computability side of the $\mathcal{X}$-calculus, we began with a review of the λ-calculus and two of its type systems as defined by Curry and by Church. Curry, Howard and de Bruijn discovered a correspondence (in fact, an isomorphism) between the λ-calculus and the Natural Deduction presentation of intuitionistic logic. It was implied by some of Griffin's work that some model of computation existed that held a similar kind of correspondence with classical logic.

The first studies into this model of computation were based on Natural Deduction formulations of Classical Logic, though Curien and Herbelin later developed a more 'natural' sequent calculus formulation. However, Curien and Herbelin's $\overline{\lambda}\mu\tilde{\mu}$-calculus did not hold a perfect correspondence with their sequent calculus for classical logic: some $\overline{\lambda}\mu\tilde{\mu}$-terms were not redexes, even though they were typed with (eliminable) the cut rule. Lengrand studied a subsystem of $\overline{\lambda}\mu\tilde{\mu}$ which restored this 'cut=redex' correspondence. The calculus he designed, called the $\lambda\xi$-calculus, preserved the symmetries of the classical sequent calculus. Studying the reduction properties of his calculus, Lengrand formulated two dual symmetric reduction subsystems which corresponded to the call-by-name and call-by-value notions of computation. The most interesting feature of these subsystems (in our opinion) was that they relied on different permutations of the principal reduction rule. The syntax of $\lambda\xi$ was later reformulated and became known as $\mathcal{X}$. Urban's work also contributed to the development of the $\mathcal{X}$-calculus.

In order to achieve a full 'Curry-Howard' style correspondence with a classical sequent calculus, the $\mathcal{X}$-calculus employs a verbose syntax. Additionally, the sixteen reduction rules (at first glance) are not immediately intuitive. The first part of our research involved studying this reduction mechanism in great detail. To this end, we sought an implementation of $\mathcal{X}$ (this was presented in Chapter 4). We based our implementation on term graph rewriting techniques, but noticing that the $\mathcal{X}$-calculus was not a simple rewrite system (most prominently it featured higher-order term constructors plus side conditions on rewrite rules), we extended the standard formulation of first order graph rewriting with features to express binding and check side conditions. Observing that a naïve implementation of $\mathcal{X}$ suffered from name clash and name capture problems, we investigated and then proposed a number of solutions. Each solution essentially performed a series of α-conversions during a reduction to maintain the variable identity and

variable binding relations encoded in the structure of the term. We quantitatively compared the operating cost of each solution using a suite a suite of benchmark terms. To ensure our results were fair, we needed to ensure the α-conversion steps introduced by each solution did not affect the reduction paths chosen. Subsequently, we extended our higher-order conditional term graph rewrite system with a strategy language (due to Visser). This language enabled us to specify a reduction strategy that essentially hid the α-conversion steps introduced by each solution from the reduction path taken by the term. The solution we called 'avoiding capture' was the most efficient, and so we internalised a generic formulation of this solution using copy nodes in the implementation of our CTGRS.

We used the tool to understand the reduction mechanism of $\mathcal{X}$. We noticed some optimisations that could be made, and presented these in Chapter 3. We also related the $\mathcal{X}$-calculus to well-understood notions of computation that employed control features, and compared its reduction mechanism to that of the $\overline{\lambda}\mu\tilde{\mu}$ calculus.

Having gained familiarity with the $\mathcal{X}$-calculus, i.e., its syntax and reduction mechanism, we turned to study the type assignment system that gave the calculus its logical foundations. The calculus is actually built on only the implicative fragment of (a variant of) the *G3a* calculus. From our investigations into other calculi with Curry-Howard correspondences for classical logic, we observed that a specific set of primitive connectives was often favoured (namely implication, conjunction, disjunction and negation). As a result, we decided to explore some of the other less well known connectives. We began with a study of the sixteen classical logical connectives of arity two. By formulating a notion of 'obtainability', we were able to group the sixteen connectives into five groups, where each connective in a group could be 'obtained' from any another connective in that group. We studied two of these groups in detail.

The first group contained what we called 'pairing' connectives, since the computational content of each of these could be related to the traditional kind of pairing functionality associated with logical conjunction. We also looked at simulating the $\mathcal{X}$-calculus in $\mathcal{X}$-style calculi built from functionally complete sets of connectives. We introduced two calculi $\mathcal{X}^{\uparrow}$ and $\mathcal{X}^{\neg\vee}$ based on the logical nand connective and the negation plus disjunction connectives respectively. We showed that even though each of these sets of connectives were functionally complete (and could therefore logically express implication), they could not (fully) computationally express the $\mathcal{X}$-calculus. In particular, each encoding could only simulate one of the principal reduction rules for $\mathcal{X}$. In other words, we showed our notion

of computational expressivity (essentially simulation) did not follow from logical expressivity.

The second group of arity two connectives we explored contained only the if-and-only-if and the exclusive or connectives. The computational content of these connectives were largely unexplored in the literate, and so we designed a calculus $\mathcal{X}^{\leftrightarrow}$ to investigate. When building the principal reduction rules for the calculus, we encountered some difficulties. First we extracted a pair of principal reduction rules by considering the cut-elimination for a sequent calculus employing logically equivalent formulation of the if-and-only-if connective. However, the right-hand side of the reduction rules copied some rule premises (the rules were not right-linear). Unsatisfied with this result, we were able to successfully construct another, simpler, set of principal reduction rules which were right-linear by considering a formulation of the rule using an intuitive diagrammatic representation. Next we looked at encoding other computational calculi in $\mathcal{X}^{\leftrightarrow}$. The if-and-only-if connective can only logically express the top connective and the identity connective, but noticing the complex structure of the 'iff input' and 'iff output' circuits, we attempted an encoding of the $\mathcal{X}$-calculus. Surprisingly, we were able to simulate one of the principal reduction rules for $\mathcal{X}$ in our chosen encoding, showing that computational expressivity was possible even though logical expressivity was not.

We were also able to present a general 'recipe' for building Curry-Howard correspondences between extensions of a specific sequent calculus with a connective, and a term calculus constructed in the style of $\mathcal{X}$. However, our first formulation of this 'recipe' did not always built the simplest form of the principal reduction rules for a logical connective. We were interested in seeing whether we could formulate an algorithm which would build the simplest principal reduction rules (i.e., the right-linear formulation). Recall that existing works might be able to do this, but only by modifying an unscalable brute-force searching procedure.

In Chapter 6, we studied the relationship between two-valued truth tables and sequent calculus style inference rules. Specifically, we found a piece of work by Call that (informally) outlined an algorithm to build a pair of invertible inference rules for a logical connective from its truth table. We spent some time formalising the exact relationship between the two structures, and gave new insight into the algorithm by relating it to the cut rule. In fact, we were able to describe a reverse algorithm which would relate the premises of the connective's inference rules with a set of rows belonging to the connective's truth tables. We introduced a structure which we called a 'bitmask' (based on a notion of three-valued truth as-

signments) that served as a succinct notation for these sets of rows. By observing the effect of applying the cut to premises of the inference rules, then relating this to the bitmasks representations of rule premises, we were able to reformulate the cut rule as an operation that worked on rows of truth tables. We were also able to give an additional geometric formulation of the cut that worked on 'hypercubes'. With this understanding, we built right-hand sides of principal reduction rules using bitmasks rather than derivation schemes. The final algorithm we presented solved this task, and in fact we were able to also specify the construction of only right-linear rules.

7.1 Future Directions

Unfortunately, due to the time requirements of the Ph.D programme, we did not have a chance to investigate all of the areas we found captivating. This section details what we consider to be the most interesting directions we would have liked to follow. Some of the work we describe in this section is currently under further investigation.

7.1.1 Investigations into *Unsimplified* Inference Rules

In 6.1.2, we broke down Call's algorithm into two steps: building a pair of inference rules, followed by the simplification of the rules. The unsimplified rules built for the implication rule were given in Example 6.1.7. We could quite easily define principal reduction rules for connectives based on the unsimplified rules; the right-hand sides would be:

$$\dfrac{\dfrac{\Gamma_1^L \vdash \Delta_1^L, A_1, A_2 \quad A_2, \Gamma_2^L \vdash \Delta_2^L, A_1}{\Gamma_1^L, \Gamma_2^L \vdash \Delta_1^L, \Delta_2^L, A_1}\,(Cut) \qquad \dfrac{A_1, \Gamma_1^R \vdash \Delta_1^R, A_2 \quad A_1, A_2, \Gamma_3^L \vdash \Delta_3^L}{A_1, \Gamma_1^R, \Gamma_3^L \vdash \Delta_1^R, \Delta_3^L}\,(Cut)}{\Gamma_1^L, \Gamma_2^L, \Gamma_3^L, \Gamma_1^R \vdash \Delta_1^L, \Delta_2^L, \Delta_3^L, \Delta_1^R}\,(Cut)$$

$$\dfrac{\dfrac{\Gamma_1^L \vdash \Delta_1^L, A_1, A_2 \quad A_1, \Gamma_1^R \vdash \Delta_1^R, A_2}{\Gamma_1^L, \Gamma_1^R \vdash \Delta_1^L, \Delta_1^R, A_2}\,(Cut) \qquad \dfrac{A_2, \Gamma_2^L \vdash \Delta_2^L, A_1 \quad A_1, A_2, \Gamma_3^L \vdash \Delta_3^L}{A_2, \Gamma_2^L, \Gamma_3^L \vdash \Delta_2^L, \Delta_3^L}\,(Cut)}{\Gamma_1^L, \Gamma_2^L, \Gamma_3^L, \Gamma_1^R \vdash \Delta_1^L, \Delta_2^L, \Delta_3^L, \Delta_1^R}\,(Cut)$$

The $\mathcal{X}$-style circuit corresponding to the unsimplified left introduction rule $(\complement^2_{1101_2} L)$

would be

$$k{\cdot}[P\widehat{\alpha}\widehat{\beta}, \widehat{z}Q\widehat{\mu}, \widehat{x}\widehat{y}R]$$

We could even extract the following pair of principal reduction rules from the above proof transformations (which are sound w.r.t. types).

$$(\widehat{w}M\widehat{\delta}{\cdot}\gamma)\widehat{\gamma}\dagger\widehat{k}(k{\cdot}[P\widehat{\alpha}\widehat{\beta}, \widehat{z}Q\widehat{\mu}, \widehat{x}\widehat{y}R]) \rightarrow ((P\widehat{\beta}\dagger\widehat{z}Q)\widehat{\alpha}\dagger\widehat{j}\langle j{\cdot}\mu\rangle)\widehat{\mu}\dagger\widehat{w}(\langle w{\cdot}\pi\rangle\widehat{\pi}\dagger\widehat{x}(M\widehat{\delta}\dagger\widehat{y}R))$$
$$(\widehat{w}M\widehat{\delta}{\cdot}\gamma)\widehat{\gamma}\dagger\widehat{k}(k{\cdot}[P\widehat{\alpha}\widehat{\beta}, \widehat{z}Q\widehat{\mu}, \widehat{x}\widehat{y}R]) \rightarrow ((P\widehat{\beta}\dagger\widehat{w}M)\widehat{\delta}\dagger\widehat{j}\langle j{\cdot}\alpha\rangle)\widehat{\alpha}\dagger\widehat{y}(\langle y{\cdot}\pi\rangle\widehat{\pi}\dagger\widehat{z}(Q\widehat{\mu}\dagger\widehat{x}R))$$

Where γ, k are introduced.

Where the contraction steps have been made explicit as cuts with capsules (as discussed in Section 5.2.3). We would have liked to study the reduction behaviour of such a computational counterparts for the logical implication connective and determine precisely the effect of simplification.

7.1.2 On the Geometry of Classical Logical Connectives

In Section 6.3, we (informally) described some relationships between the sequent calculus inference rules for a logical connective and hypercube graphs. We suggest some directions in which this work could be taken.

First, we would have liked to spent time formalising the precise relationship between the inference rules for a connective and its geometrical representation.

Having *re*discovered the relationship between Boolean functions of n logical variables and 2-colourings of the n-cube, we were made aware (by [12]) that considering equivalence classes of logical connectives (as we did in Section 5.3 with our notion of 'obtainability') was studied as far back as the 1800s. Jevons mapped the problem onto 2-colourings of 2-cubes and 3-cubes [51]. Other methods employ Burnside-P'olya Counting theory and computational group theory to count the unique equivalence classes of logical connectives of a particular arity.

In some preliminary investigations, we enumerated the class of arity-three connectives (which has 256 connectives). We grouped these connectives based on the structure of the right-hand sides of cut-elimination rules and discovered there were 14 unique groups (based on our grouping criteria). We remark that there are also 14 (or 15) equivalence classes of 3-cubes, depending on which relations are used to build the equivalence class. Whether a relationship between the two exists is a topic we are currently researching.

We would have also liked to extend our main algorithm `solutions` to full generality to work on k-colourings of n-cubes, rather than just 2-colourings; this is also

a problem studied in [12]. We hope that this would generalise our algorithm to the setting of many-valued logics.

7.1.3 On the Computation Content of the Cross-Cut

In the $\mathcal{X}$-style calculi we studied in this thesis, contractions are encoded with a cut and a capsule (see Section 5.2). It is fairly straightforward to extend our sequent calculus with explicit rules for contraction and corresponding $\mathcal{X}$-style circuits. In fact, Lescanne and Žunič studied $\mathcal{X}$-style calculi with explicit circuit representations for contraction in their investigations into the computational content of linear logic [90, 63]. Borrowing their term annotations, we could extend the set of circuits with the following circuit-constructors which inhabit the left and right inference rules for contraction:

$$P, Q ::= \ldots \mid \underbrace{[P\rangle^{\hat{\alpha}}_{\hat{\beta}}{>}\gamma}_{\text{Contraction Output}} \mid \underbrace{z{<}^{\hat{y}}_{\hat{x}}\langle Q]}_{\text{Contraction Input}}$$

Observe that we can encode these circuits for explicit contraction rules into the $\mathcal{X}$-style calculi we formulated in this thesis:

$$\begin{aligned} \llbracket [P\rangle^{\hat{\alpha}}_{\hat{\beta}}{>}\gamma \rrbracket &= (P\widehat{\alpha} \dagger \widehat{i}\langle j{\cdot}\gamma\rangle)\widehat{\beta} \dagger \widehat{j}\langle j{\cdot}\gamma\rangle && i, j \textit{ fresh} \\ \llbracket z{<}^{\hat{y}}_{\hat{x}}\langle Q] \rrbracket &= \langle z{\cdot}\pi\rangle\widehat{\pi} \dagger \widehat{x}(\langle z{\cdot}\sigma\rangle\widehat{\sigma} \dagger \widehat{y}Q) && \pi, \sigma \textit{ fresh} \end{aligned}$$

In determining how to reduce a cut built using the above circuits, we first remind ourselves that the sequent calculus rules for contraction are 'structural rules' and not logical rules. One very important difference is we can build a cut whose cut formula is *introduced* by different inference rules, for example,

$$\dfrac{\dfrac{}{\mathfrak{a} \vdash \mathfrak{a}}\,(Ax)}{\vdash \mathfrak{a}\rightarrow\mathfrak{a}}\,(\rightarrow R) \qquad \dfrac{\dfrac{\dfrac{}{\mathfrak{a}\rightarrow\mathfrak{a}, \mathfrak{a} \vdash \mathfrak{a}}\,(Ax) \qquad \dfrac{}{\mathfrak{a} \vdash \mathfrak{a}}\,(Ax)}{\mathfrak{a}\rightarrow\mathfrak{a}, \mathfrak{a}\rightarrow\mathfrak{a} \vdash \mathfrak{a}}\,(\rightarrow L)}{\mathfrak{a}\rightarrow\mathfrak{a} \vdash \mathfrak{a}}\,(\textit{ContractionL})}{\vdash \mathfrak{a}}\,(\textit{Cut})$$

A witness for the above proof is:

$$(\widehat{w}\langle w{\cdot}\beta\rangle\widehat{\beta}{\cdot}\alpha)\widehat{\alpha} \dagger \widehat{z}(z{<}^{\hat{y}}_{\hat{x}}\langle(\langle u{\cdot}\pi\rangle\widehat{\pi}\,[y]\,\widehat{v}\langle v{\cdot}\sigma\rangle)]\rangle)$$

In the above, the cut formula is introduced by the right premise of the cut; the corresponding socket z is also introduced. In $\mathcal{X}$-style calculi, when both connectors

of the cut are introduced by the respective rules, a logical rule would be applied. When logical connectives are concerned, usually the 'principal reduction rule' we studied is applicable, which upon application would reduce the complexity of the associated cut formula.

Where contractions are concerned, the notion of 'principal reduction rule' *in general* does not make sense, since there is no requirement to introduce the cut formula using both a left and a right contraction rule. Keeping this in mind when formulating an appropriate proof transformation rule, the right-hand side actually becomes quite simple. We could add the following reduction rules, which simply copy the sub-circuit of the cut that is in interaction with the contraction circuit:

$$\begin{array}{lllll}
(ContractR_1): & P\widehat{\alpha}\dagger\widehat{z}(z{<}^{\widehat{y}}_{\widehat{x}}\langle Q]) & \rightarrow P\widehat{\alpha}\dagger\widehat{x}(P\widehat{\alpha}\dagger\widehat{y}Q) & \leftarrow z \text{ introduced} \\
(ContractR_2): & P\widehat{\alpha}\dagger\widehat{z}(z{<}^{\widehat{y}}_{\widehat{x}}\langle Q]) & \rightarrow P\widehat{\alpha}\dagger\widehat{y}(P\widehat{\alpha}\dagger\widehat{x}Q) & \leftarrow z \text{ introduced} \\
(ContractL_1): & ([P\rangle^{\hat{\alpha}}_{\hat{\beta}}{>}\gamma)\widehat{\gamma}\dagger\widehat{z}Q & \rightarrow (P\widehat{\alpha}\dagger\widehat{z}Q)\widehat{\beta}\dagger\widehat{z}Q & \leftarrow \gamma \text{ introduced} \\
(ContractL_2): & ([P\rangle^{\hat{\alpha}}_{\hat{\beta}}{>}\gamma)\widehat{\gamma}\dagger\widehat{z}Q & \rightarrow (P\widehat{\beta}\dagger\widehat{z}Q)\widehat{\alpha}\dagger\widehat{z}Q & \leftarrow \gamma \text{ introduced}
\end{array}$$

Notice that in the above, we would expect the appropriate connector to be introduced since, after all, the sequent calculus now features explicit rules for contraction. By inspection, one can easily verify that the addition of the above reduction rules will preserve the 'cut=redex' paradigm in $\mathcal{X}$-style calculi, and (by extending the cut-elimination with the corresponding proof transformation rules) also preserve the cut-elimination property of an extension of our sequent calculus with explicit rules for contraction.

In the (very specific) case where both connectors of a cut are introduced by contraction circuits, the reduction becomes very complex, i.e.,

$$([P\rangle^{\hat{\alpha}}_{\hat{\beta}}{>}\gamma)\widehat{\gamma}\dagger\widehat{z}(z{<}^{\widehat{y}}_{\widehat{x}}\langle Q]) \tag{7.1}$$

This reduces as follows,

$$\begin{array}{lll}
\text{Circuit (7.1)} = & ([P\rangle^{\hat{\alpha}}_{\hat{\beta}}{>}\gamma)\widehat{\gamma}\dagger\widehat{z}(z{<}^{\widehat{y}}_{\widehat{x}}\langle Q]) & \rightarrow (ContractR_1) \\
& ([P\rangle^{\hat{\alpha}}_{\hat{\beta}}{>}\gamma)\widehat{\gamma}\dagger\widehat{x}(([P\rangle^{\hat{\alpha}}_{\hat{\beta}}{>}\gamma)\widehat{\gamma}\dagger\widehat{y}Q) & \rightarrow (ContractL_1) \\
& ([P\rangle^{\hat{\alpha}}_{\hat{\beta}}{>}\gamma)\widehat{\gamma}\dagger\widehat{x}((P\widehat{\alpha}\dagger\widehat{y}Q)\widehat{\beta}\dagger\widehat{y}Q) & \rightarrow (ContractL_1) \\
& (P\widehat{\alpha}\dagger\widehat{x}((P\widehat{\alpha}\dagger\widehat{y}Q)\widehat{\beta}\dagger\widehat{y}Q))\widehat{\beta}\dagger\widehat{x}((P\widehat{\alpha}\dagger\widehat{y}Q)\widehat{\beta}\dagger\widehat{y}Q) &
\end{array}$$

The decomposition of the contraction circuits creates a circuit with six cuts, copying one sub-circuit three times, and the other four times.

Gentzen defined an additional proof transformation that acted on contractions

in the very specific case described above shown in Circuit (7.1); he called this transformation *cross-cut*. We slightly modify his formulation in our presentation of the rule shown below.

$$\dfrac{\dfrac{\boxed{D_R}}{\Gamma \vdash \Delta, A, A}}{\Gamma \vdash \Delta, A}\,(Contraction_R) \qquad \dfrac{\dfrac{\boxed{D_L}}{A, A, \Gamma \vdash \Delta}}{A, \Gamma \vdash \Delta}\,(Contraction_L)}{\Gamma \vdash \Delta}\,(Cut)$$

reduces to:

$$\dfrac{\dfrac{\boxed{D_R}}{\Gamma \vdash \Delta, A, A} \quad \dfrac{\boxed{D_L}}{A, A, \Gamma \vdash \Delta}}{\Gamma \vdash \Delta, A}\,(Contraction_L) \qquad \dfrac{\dfrac{\boxed{D_R}}{\Gamma \vdash \Delta, A, A}}{\Gamma \vdash \Delta, A}\,(Contraction_R) \quad \dfrac{\boxed{D_L}}{A, A, \Gamma \vdash \Delta}}{A, \Gamma \vdash \Delta}\,(Cut)}{\Gamma \vdash \Delta}\,(Cut)$$

In this case, the cut is formed where the cut formula is introduced by both a right and left contraction rule. We remark that this is reminiscent of a 'principal reduction rule'.

We could extract an $\mathcal{X}$-style reduction rule based on the above 'cross-cut' transformation; this is shown below.

$$(\textit{cross-cut}_1): \quad ([P\rangle^{\hat{\alpha}}_{\hat{\beta}} > \gamma)\widehat{\gamma} \dagger \widehat{z}(z < ^{\hat{y}}_{\hat{x}} \langle Q]) \rightarrow (P\widehat{\alpha} \dagger \widehat{z}(z < ^{\hat{y}}_{\hat{x}} \langle Q]))\widehat{\beta} \dagger \widehat{x}(([P\rangle^{\hat{\alpha}}_{\hat{\beta}} > \gamma)\widehat{\gamma} \dagger \widehat{y}Q)$$

Note that there are three other permutations of the above right-hand side.

Now we could reduce Circuit (7.1) by first applying this rule then proceeding using the previously defined rules for contraction, i.e.:

$$\begin{aligned}
\text{Circuit (7.1)} = \; & ([P\rangle^{\hat{\alpha}}_{\hat{\beta}} > \gamma)\widehat{\gamma} \dagger \widehat{z}(z < ^{\hat{y}}_{\hat{x}} \langle Q]) && \rightarrow (\textit{cross-cut}_1) \\
& (P\widehat{\alpha} \dagger \widehat{z}(z < ^{\hat{y}}_{\hat{x}} \langle Q]))\widehat{\beta} \dagger \widehat{x}(([P\rangle^{\hat{\alpha}}_{\hat{\beta}} > \gamma)\widehat{\gamma} \dagger \widehat{y}Q) && \rightarrow (\textit{ContractR}_1) \\
& (P\widehat{\alpha} \dagger \widehat{x}(P\widehat{\alpha} \dagger \widehat{y}Q))\widehat{\beta} \dagger \widehat{x}(([P\rangle^{\hat{\alpha}}_{\hat{\beta}} > \gamma)\widehat{\gamma} \dagger \widehat{y}Q) && \rightarrow (\textit{ContractL}_1) \\
& (P\widehat{\alpha} \dagger \widehat{x}(P\widehat{\alpha} \dagger \widehat{y}Q))\widehat{\beta} \dagger \widehat{x}((P\widehat{\alpha} \dagger \widehat{z}Q)\widehat{\beta} \dagger \widehat{z}Q)
\end{aligned}$$

Using the $(\textit{cross-cut}_1)$ rule appears to be a kind of optimisation. Notice that the above circuit has one less cut than using the separate contraction rules. Also, the reduct makes one less copy of a sub-circuit.

We spent some time looking at reductions involving contraction circuits in $\mathcal{X}$, and in particular, observing if the possibility to apply the cross-cut rule arose during the reduction of a 'typical' circuit (i.e., one which was not formulated especially

so that the cross-cut rule would be applicable). With the arity two connectives we studied, we did not encounter any such instances in our investigations. However, we did notice that in some situations a 'hybrid' cross-cut rule would be applicable. In our sequent calculus (which has sets of labelled formulas), we would usually formulate contraction as:

$$\frac{x{:}A, y{:}A, \Gamma \vdash \Delta}{z{:}A, \Gamma \vdash \Delta}\,(\mathit{ContractL}) \qquad \frac{\Gamma \vdash \Delta, \alpha{:}A, \beta{:}A}{\Gamma \vdash \Delta, \gamma{:}A}\,(\mathit{ContractR})$$

However, by suitably choosing the label of the introduced formula (in the conclusion of the rule), we could have instead used the following formulation:

$$\frac{x{:}A, y{:}A, \Gamma \vdash \Delta}{x{:}A, \Gamma \vdash \Delta}\,(\mathit{HybridContractL}) \qquad \frac{\Gamma \vdash \Delta, \alpha{:}A, \beta{:}A}{\Gamma \vdash \Delta, \alpha{:}A}\,(\mathit{HybridContractR})$$

The term annotations for these circuits would be $P\widehat{\beta}\succ\!\alpha$ and $x\!\prec\!\widehat{y}Q$ respectively. We noticed we could add the following reduction rule, in the spirit of the cross-cut.

$$(\mathit{HybridCC}_1)\colon \quad (P\widehat{\beta}\succ\!\alpha)\widehat{\alpha} \dagger \widehat{x}(x\!\prec\!\widehat{y}Q) \;\rightarrow\; (P\widehat{\beta} \dagger \widehat{x}(x\!\prec\!\widehat{y}Q))\widehat{\alpha} \dagger \widehat{x}((P\widehat{\beta}\succ\!\alpha)\widehat{\alpha} \dagger \widehat{y}Q)$$

Note that in the above reduction, α and x might not be introduced.

Now, using the following encoding,

$$\begin{aligned} [\![x\!\prec\!\widehat{y}Q]\!] &= \langle x{\cdot}\pi\rangle\widehat{\pi} \dagger \widehat{y}Q \\ [\![P\widehat{\beta}\succ\!\alpha]\!] &= P\widehat{\beta} \dagger \widehat{k}\langle k{\cdot}\alpha\rangle \end{aligned}$$

we could build a reduction rule $(\mathit{HybridCC}'_1)$:

$$(P\widehat{\beta} \dagger \widehat{k}\langle k{\cdot}\alpha\rangle)\widehat{\alpha} \dagger \widehat{x}(\langle x{\cdot}\pi\rangle\widehat{\pi} \dagger \widehat{y}Q) \;\rightarrow\; (P\widehat{\beta} \dagger \widehat{x}(\langle x{\cdot}\pi\rangle\widehat{\pi} \dagger \widehat{y}Q))\widehat{\alpha} \dagger \widehat{x}((P\widehat{\beta} \dagger \widehat{k}\langle k{\cdot}\alpha\rangle)\widehat{\alpha} \dagger \widehat{y}Q)$$

The above rule uses the cut with a capsule to simulate contraction. One should bear in mind that there may be occurrences of α in P, and occurrences of x in Q.

The above rule is applicable to the right-hand side of both principal reduction rules in the $\mathcal{X}^{\leftrightarrow}$-calculus[1]. We recall the rule, $(\leftrightarrow_1)$ below (from Figure 5.8).

$$(\leftrightarrow_1)\colon \quad ((M\widehat{\mu} \dagger \widehat{x}P)\widehat{\sigma} \dagger \widehat{k}\langle k{\cdot}\alpha\rangle)\widehat{\boldsymbol{\alpha}} \dagger \widehat{\boldsymbol{z}}(\langle z{\cdot}\pi\rangle\widehat{\pi} \dagger \widehat{j}(Q\widehat{\delta} \dagger \widehat{i}N))$$

The highlighted cut, $\widehat{\alpha} \dagger \widehat{z}$ by applying either $(\mathit{act}\text{-L})$ or $(\mathit{act}\text{-R})$ as usual then propagating appropriately, but also by the rule $(\mathit{HybridCC}'_1)$. An investigation into the exact gains from this optimisation is left open as a possibility for future work.

[1] We would like to thank Herbelin who first brought the cross-cut proof transformation to our attention. In fact, at a workshop in Vienna (DCM 2006), he even remarked our calculus $\mathcal{X}^{\leftrightarrow}$ might be related to Gentzen's cross-cut.

Work on the cross-cut in the literature is sparse; we only managed to find a few citations (of which we found [40] particularly readable). It appears to us that the main reason the cross-cut has not been studied in a Curry-Howard setting is mainly because it has been difficult to formulate a suitable term calculus whose syntax can express contraction, and reduction rules can embody the cross-cut transformation itself. Since $\mathcal{X}$ is a straightforward annotation for sequent calculus proofs, we consider it to be an ideal setting for further investigations.

We also note that (in unpublished work) Herbelin has also considered extending the $\overline{\lambda}\mu\tilde{\mu}$-calculus with a cross-cut reduction rule. He hints it could lead to another reduction paradigm (i.e., call-by-name vs. call-by-value vs. cross-cut).

Bibliography

[1] Zena Ariola, Aaron Bohannon, and Amr Sabry. Sequent calculi and abstract machines. To appear in the ACM Transactions on Programming Languages and Systems., February 2005.

[2] Zena Ariola and Hugo Herbelin. Minimal classical logic and control operators.

[3] Zena M. Ariola, Hugo Herbelin, and Amr Sabry. A proof-theoretic foundation of abortive continuations (extended version). Technical report, Indiana University, 2005.

[4] Andrea Asperti, Cecilia Giovanetti, and Andrea Naletto. The bologna optimal higher-order machine. *J. Funct. Program.*, 6(6):763–810, 1996.

[5] Matthias Baaz and Christian G. Fermüller. Resolution for many-valued logics. In *LPAR '92: Proceedings of the International Conference on Logic Programming and Automated Reasoning*, pages 107–118, London, UK, 1992. Springer-Verlag.

[6] Matthias Baaz, Christian G. Fermüller, Arie Ovrutcki, and Richard Zach. Multilog: A system for axiomatizing many-valued logics. In *LPAR '93: Proceedings of the 4th International Conference on Logic Programming and Automated Reasoning*, pages 345–347, London, UK, 1993. Springer-Verlag.

[7] Matthias Baaz, Christian G. Fermüller, and Richard Zach. Elimination of cuts in first-order finite-valued logics. *Elektronische Informationsverarbeitung und Kybernetik*, 29(6):333–355, 1993.

[8] S. van Bakel, L. Cardelli, and M.G. Vigliotti. From $\mathcal{X}$ to πi: Representing classical sequent calculus in π-calculus. In *International Workshop on Classical Logic and Computation (CL&C'08), Reykjavik, Iceland*, July 2008.

[9] S. van Bakel, S. Lengrand, and P. Lescanne. The language $\mathcal{X}$: circuits, computations and classical logic. In Mario Coppo, Elena Lodi, and G. Michele Pinna, editors, *Proceedings of Ninth Italian Conference on Theoretical Computer Science (ICTCS'05), Siena, Italy*, volume 3701 of *Lecture Notes in Computer Science*, pages 81–96. Springer-Verlag, 2005.

[10] S. van Bakel and J. Raghunandan. Implementing $\mathcal{X}$. In *Electronic Proceedings of Second International Workshop on* Term Graph Rewriting 2004 *(TermGraph'04), Rome, Italy*, Electronic Notes in Theoretical Computer Science, 2005.

[11] Steffen van Bakel. Type systems for programming languages (course notes), August 2006.

[12] David C. Banks, Stephen A. Linton, and Paul K. Stockmeyer. Counting cases in substitope algorithms. *IEEE Trans. Vis. Comput. Graph.*, 10(4):371–384, 2004.

[13] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, New York, 1981.

[14] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In *Proceedings of PARLE, Parallel Architectures and Languages Europe,* Eindhoven, The Netherlands, volume 259-II of *Lecture Notes in Computer Science*, pages 141–158. Springer-Verlag, 1987.

[15] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Towards an Intermediate Language based on Graph Rewriting. In *Proceedings of PARLE, Parallel Architectures and Languages Europe,* Eindhoven, The Netherlands, volume 259-II of *Lecture Notes in Computer Science*, pages 159–175. Springer-Verlag, 1987.

[16] Erik Barendsen and Sjaak Smetsers. Extending graph rewriting with copying. In *Dagstuhl Seminar on Graph Transformations in Computer Science*, pages 51–70, 1993.

[17] Nuel D. Belnap. Tonk, plonk and plink. *Analysis*, 22(6):130–134, 1962.

[18] R. Bloo and K.H. Rose. Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. In *CSN95 – Computer Science in the Netherlands*, pages 62–72, 1995.

[19] Eduardo Bonelli, Delia Kesner, and Alejandro Rios. A de bruijn notation for higher-order rewriting. In *RTA*, pages 62–79, 2000.

[20] Denis Bonnay and Benjamin Simmenauer. Tonk strikes back. *The Australasian Journal of Logic*, 3:33–44, 2005.

[21] Nicolas Bourbaki. *Eléments de Mathématique XXII: Théories des Ensembles, Livre I, Structures*. Number 1258 in Actualités scientifiques et industrielles. Hermann, 1957.

[22] N. G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Proc. of Symposium on Automatic Demonstration, Versailles, France, Dec. 1968*, volume 125 of *LNM 125*, pages 29–61. Springer-Verlag, Berlin, 1970.

[23] N. G. de Bruijn. Lambda calculus notation with nameless dummies. a tool for automatic formula manipulation with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[24] Richard L. Call. Constructing sequent rules for generalized propositional logics. *Notre Dame Journal of Formal Logic*, 25(2):171–178, 1984.

[25] Felice Cardone and J. Roger Hindley. The history of lambda-calculus and combinatory logic. *Handbook of the History of Logic*, 5, 2006. To appear.

[26] Serenella Cerrito and Delia Kesner. Pattern matching as cut elimination. In *Logic in Computer Science*, pages 98–108, 1999.

[27] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[28] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[29] Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 3:472–482, 1936.

[30] Agata Ciabattoni and Alexander Leitsch. Towards an algorithmic construction of cut-elimination procedures†. *Mathematical. Structures in Comp. Sci.*, 18(1):81–105, 2008.

[31] Tristan Crolard. A formulae-as-types interpretation of subtractive logic. *J. Log. Comput.*, 14(4):529–570, 2004.

[32] Pierre-Louis Curien and Hugo Herbelin. The Duality of Computation. In *Proceedings of the 5 th ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 233–243. ACM, 2000.

[33] Haskell B. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland, 1958.

[34] Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. A new deconstructive logic: Linear logic. 1995.

[35] Arie van Deursen and Joost Visser. Source model analysis using the JJ-Traveler visitor combinator framework. *Softw. Pract. Exper.*, 35(4):1345–1379, 2005.

[36] Michael Dummett. *The Logical Basis of Metaphysics*. Harvard University Press, Cambridege, MA, 1991.

[37] M. Felleisen, D. Friedman, E. Kohlbecker, and B. Duba. Reasoning with continuations. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 131–141, 1986.

[38] Mattias Felleisen. The theory and practice of first-class prompts. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 180–190, New York, NY, USA, 1988. ACM.

[39] Michael Gabbay. We can be in harmony with classical logic. Submitted for review, September 2007.

[40] J. Gallier. Constructive logic part ii: Linear logic and proof nets, 1991.

[41] Gerhard Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1934. Translated in Sabo (ed.), *The Collected Papers of Gerhard Gentzen* as "Investigations into Logical Deduction".

[42] John J. Glauert, Delia Kesner, and Zurab Khasidashvili. Expression reduction systems and extensions: An overview. In Aart Middeldorp, Vincent van Oostrom, Femke van Raamsdonk, and Roel C. de Vrijer, editors, *Processes, Terms and Cycles*, volume 3838 of *Lecture Notes in Computer Science*, pages 496–553. Springer, 2005.

[43] Timothy G. Griffin. A formulae-as-type notion of control. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–58, New York, NY, USA, 1990. ACM.

[44] Philippe de Groote. Strong normalization of classical natural deduction with disjunction. In *TLCA*, pages 182–196, 2001.

[45] Ian Hacking. What is logic? *The Journal of Philosophy*, 76:285–319, 1979.

[46] H. Herbelin. calculus structure isomorphic to gentzen-style sequent calculus structure; lncs 933, 1995.

[47] Hugo Herbelin. *C'est maintenant qu'on calcule, au cœur de la dualité*. PhD thesis, Université Paris 11, December 2005.

[48] Hugo Herbelin. Duality of computation and sequent calculus: a few more remarks. Unpublished manuscript., January 2008.

[49] W. A. Howard. The Formulae-As-Types Notion Of Construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, Inc., New York, N.Y., 1980.

[50] Von Plato J. Rereading gentzen. *Synthese*, 137:195–209(15), November 2003.

[51] W. S. Jevons. Solutions of the inverse or inductive problem, involving two classes. *Proc. Manchester Literary and Philosophical Soc.*, xi:65–68, December 1871.

[52] W. Kahl. Relational treatment of term graphs with bound variables, 1997.

[53] Wolfram Kahl. *Algebraische Termgraphersetzung mit gebundenen Variablen*. Reihe Informatik. Herbert Utz Verlag Wissenschaft, München, 1996. ISBN 3-931327-60-4; also doctoral dissertation at Fakultät für Informatik, Universität der Bundeswehr München.

[54] Wolfram Kahl and Claudia Hattensperger. Second-order syntax in hops and in ralf.

[55] J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J. de Vries. *The adequacy of term graph rewriting for simulating term rewriting*, chapter Chapter, pages 157–169. John Wiley and Sons Ltd., Chichester, UK, 1993.

[56] Delia Kesner, Laurence Puel, and Val Tannen. A typed pattern calculus. *Inf. Comput.*, 124(1):32–61, 1996.

[57] Oiva Ketonen. Untersuchungen zum pradikatenkalkul. *Annales Academiae Scientiarium Fennicae*, Series A.I. Mathematicica-physica(23):77, 1944.

[58] S. C. Kleene. *Introduction to Metamathematics*, volume 1 of *Bibliotheca Mathematica*. Van Nostrand, Amsterdam, 1952.

[59] S.C. Kleene. *Mathematical Logic*. John Wiley, New York, 1967.

[60] Jan Willem Klop. *Combinatory Reduction Systems*. PhD thesis, Utrecht University, Amsterdam, 1980. CWI Tract 127.

[61] S. Lengrand. A computational interpretation of the *cut*-rule in classical sequent calculus. Master's thesis, Mathematical Institute & Computing Laboratory, University of Oxford, 2002.

[62] Stéphane Lengrand. Call-by-value, call-by-name, and strong normalization for the classical sequent calculus. In Bernhard Gramlich and Salvador Lucas, editors, *Post-proceedings of the 3rd Workshop on Reduction Strategies in Rewriting and Programming (WRS 2003)*, volume 86 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[63] Pierre Lescanne and Dragiša Žunič. Rewriting diagrams for computing and interpreting classical logic. *19th International Workshop on Algebraic Development Techniques*, 2008.

[64] Paul Blain Levy. Jumbo *ambda*-calculus. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *ICALP (2)*, volume 4052 of *Lecture Notes in Computer Science*, pages 444–455. Springer, 2006.

[65] Ian Mackie. Efficient lambda-evaluation with interaction nets. In *RTA*, pages 155–169, 2004.

[66] Conor McBride and James McKinna. Functional pearl: I am not a number; i am a free variable. In *Proceedings of the 2004 ACM SIGPLAN Haskell Workshop*, pages 1–9, New York, 2004. ACM Press.

[67] C.-H. L. Ong and C. A. Stewart. A curry-howard foundation for functional computation with control. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 215–227, New York, NY, USA, 1997. ACM.

[68] M. Parigot. An algorithmic interpretation of classical natural deduction. In *Proc. of Int. Conf. on Logic Programming and Automated Reasoning, LPAR'92*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer-Verlag, 1992.

[69] Michel Parigot. Free deduction: An analysis of "computations" in classical logic. In Andrei Voronkov, editor, *RCLP*, volume 592 of *Lecture Notes in Computer Science*, pages 361–380. Springer, 1991.

[70] Gordon D. Plotkin. Call-by-name, call-by-value and the [lambda]-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[71] Dag Prawitz. *Natural Deduction, A Proof-Theoretical Study*. Almqvist & Wiksell, 1965.

[72] Arthur N. Prior. The runabout inference-ticket. *Analysis*, 21:38–39, 1960.

[73] Femke van Raamsdonk. *Confluence and Normalization for Higher-Order Rewriting*. PhD thesis, Vrije Universiteit, Amsterdam, 1996.

[74] Jayshan Raghunandan and Alexander J. Summers. On the computational representation of classical logical connectives. *Electr. Notes Theor. Comput. Sci.*, 171(3):85–109, 2007.

[75] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3–4):233–247, 1993.

[76] Bertrand Russell and Alfred North Whitehead. *Principia Mathematica*. Cambridge University Press, Cambridge, 1910.

[77] Read S. Harmony and autonomy in classical logic. *Journal of Philosophical Logic*, 29:123–154(32), April 2000.

[78] Peter Selinger. Control categories and duality: On the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(2):207–260, 2001.

[79] Peter Sestoft. Demonstrating lambda calculus reduction, 2002.

[80] R. Sleep, M.J. Plasmeijer, and M.C.J.C van Eekelen, editors. *Term Graph Rewriting. Theory and Practice*. Wiley, 1993.

[81] Mark-Oliver Stehr. Cinni - a generic calculus of explicit substitutions and its application to lambda-, varsigma- and pi- calculi. *Electr. Notes Theor. Comput. Sci.*, 36, 2000.

[82] Alexander J. Summers. Interpretation of λ-calculus terms to $\mathcal{X}$ according to Prawitz's natural deduction to sequent calculus translation. A personal communication, June 2007.

[83] Alexander J. Summers. A curry-howard correspondence for a canonical classical natural deduction: Extending $\lambda\mu$ with first-class continuations. In *International Workshop on Classical Logic and Computation (CL&C'08), Reykjavik, Iceland*, July 2008.

[84] Alexander J. Summers. *Curry-Howard Term Calculi for Gentzen-Style Classical Logics*. PhD thesis, Imperial College London, October 2008.

[85] Alexander J. Summers and Steffen van Bakel. Approaches to polymorphism in classical sequent calculus. In Peter Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2006.

[86] Christian Urban. *Classical Logic and Computation*. PhD thesis, University of Cambridge, October 2000.

[87] E. Visser and Z. Benaissa. A core language for rewriting, 1998.

[88] Eelco Visser. The stratego tutorial.

[89] Joost Visser. Visitor combination and traversal control. In *Conference on Object-Oriented*, pages 270–282, 2001.

[90] Dragiša Žunič. *Computing with Sequents and Diagrams in Classical Logic - Calculi* ${}^{*}\mathcal{X}$, ${}^{d}\mathcal{X}$ *and* ${}^{\copyright}\mathcal{X}$. PhD thesis, Ecole Normale Supérieure de Lyon, 2007.

[91] Philip Wadler. Call-by-value is dual to call-by-name - reloaded. In Jürgen Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 185–203. Springer, 2005.

[92] Christopher Peter Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, Programming Research Group, University of Oxford, September 1971.