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

Curry-Howard Term Calculi for Gentzen-Style Classical Logics

Alexander J. Summers

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, August 2008

The content of this thesis is all my own work. This thesis has been typeset in LaTeX with the aid of macros written by Paul Taylor, Steffen van Bakel and Jayshan Raghunandan.

Signed:

Alexander Summers
15th August 2008

Abstract

This thesis is concerned with the extension of the Curry-Howard Correspondence to classical logic. Although much progress has been made in this area since the seminal paper by Griffin, we believe that the question of finding canonical calculi corresponding to classical logics has not yet been resolved. We examine computational interpretations of classical logics which we keep as close as possible to Gentzen's original systems, equipped with general notions of reduction.

We present a calculus $\mathcal{X}^i$ which is based on classical sequent calculus and the strongly-normalising cut-elimination procedure defined by Christian Urban. We examine how the notion of shallow polymorphism *à la* ML can be adapted to this calculus. We show that the intuitive adaptation of these ideas fails to be sound, and give a novel solution.

In the setting of classical natural deduction, we examine the $\lambda\mu$-calculus of Parigot. We show that the underlying logic is incomplete in various ways, compared with a standard Gentzen-style presentation of classical natural deduction. We relax the identified restrictions, yielding a richer calculus ($\nu\lambda\mu$) with a new kind of binding explicitly representing first-class continuations.

We examine the relationship between various existing control operators in the literature and the $\nu\lambda\mu$-calculus. We show that the μ-binding, along with our generalised reduction rules, performs the role of a delimited control operator.

We define a mapping from $\mathcal{X}^i$ to $\nu\lambda\mu$, preserving typings and reductions. We believe this is the first time a general notion of cut-elimination for classical sequent calculus has been encoded into a calculus based on a Gentzen-style presentation of classical natural deduction. This encoding allows various of our results from the previous chapters to be adapted from one paradigm to the other.

Acknowledgements

This thesis has been inspired, motivated and aided by a great many people, all of whom I am grateful to, and some of whom I will undoubtedly neglect to mention below.

My first supervisor has been Steffen van Bakel, whose course and teaching first inspired my interest in type systems, and whose belief in and encouragement of my research has been steadfast throughout. We have had many very long and fruitful discussions, without which the technical and personal achievements made here would not have been possible. I have been very fortunate to have Luca Cardelli as my second supervisor, who has been willing to find time for many enlightening and enjoyable discussions on this work. Thanks also to my examiners Gavin Bierman and Hugo Herbelin, for a thorough and interesting examination and also for encouraging me to subsequently publish my PhD work.

I am extremely grateful to Jayshan Raghunandan, who has been my PhD comrade throughout, listening patiently to me ranting about all kinds of things which I found exciting or distressing at the time, working through many research problems with me, being excellent company on various trips away, and generally being invaluable to my day-to-day work. I also have been lucky to have an excellent collection of officemates, and would like to thank Billiejoe, Ioana, Jasp and Simon especially, for being prepared to discuss practically anything and for keeping me amused during the long college hours.

A number of my other friends have shown an interest in my work, and still more have supported me through other means (particularly tea). I am especially grateful to Dorian, Jamie, Tom, Sam and Jo, Naomi, Doug, Mary, Luke, Rob and the other DeIViants, Harry, Fi, Myra, David, Ant and Martin, and I'm sure many others who should be included too. My godparents Basia and Anthony have provided me with many excellent meals and evenings, and given me much advice, all of which I am very grateful for.

Sophia Drossopoulou has been kind enough to show a great interest, kindness and support regarding my work. She has also provided me with useful advice and feedback, as well as interesting further work together. I am also grateful to Susan Eisenbach for devoting several long meetings to discussing both my PhD work and my future plans, and for additional support afterwards. Within our department I received great help and encouragement from many other academics, particularly Krysia Broda, Gabrielle Sinnadurai and Ian Hodkinson. Thanks also to the SLURP research group, particularly Nick Cameron, for welcoming me. I am very grateful to David Allman, whose support, understanding and encouragement have been essential to me successfully completing this work.

During my PhD the following people have, at different times, shown a generous interest in my work and made valuable suggestions: Maria-Grazia Vigliotti, Pierre Lescanne, Dragisa Zunic, Stephane Lengrand, Hugo Herbelin, Joe Wells, Mariangiola Dezani, Christian Urban, Philip Wadler, Ulrich Berger, Helmut Schwichtenberg, Jan von Plato.

Most importantly, I am very grateful to my family for their consistent support; both emotional and practical. I am very lucky to have so many loving relatives, and hope to spend more time with them now that my thesis is finished. This work is dedicated particularly to my grandmother Janet Small, grandfather Ken Barker and my friends John Cole and Gabrielle Sinnadurai, who would all have been very proud to see me complete it.

Contents

Chapter 1

Introduction

Since the days of Gödel, Curry, Church, Turing and Shannon, the foundations of computer science have always been intrinsically linked with mathematics, and particularly mathematical logic. All of the founders of computer science have been mathematicians, and many were also logicians. The historic link between mathematics and computer science was strengthened by the discovery of the Curry-Howard Correspondence [22, 23, 50], providing a direct connection between the fields of functional programming and formal logic. In the past two decades, interest in this correspondence has been rejuvenated by the observation that the typical isomorphism between purely functional languages and minimal logics might be extended to relate programming calculi incorporating more-expressive features (control operators) with classical logics. A wealth of research has followed, ranging from practical attempts to understand existing programming disciplines in a logical sense, to foundational approaches concerning the essences of these two subjects.

The mathematician and logician Gerhard Gentzen, although unconcerned himself with computer science, was the inventor of the two systems of formal logic most commonly used ever since, being natural deduction calculi and sequent calculi [39]. As a result of the Curry-Howard Correspondence his work has become significant for the foundations of computer science, particularly functional programming, since it is a natural deduction formulation of minimal logic which famously corresponds with Church's λ-calculus [20]. This original correspondence is very clean: neither the logic or the programming calculus need be changed in order to obtain a full-isomorphism of reductions as well as syntactic entities (proofs-as-programs, formulas-as-types).

Regarding possible isomorphisms between programming calculi and classical logic, it has been less clear what the best approach might be. While some authors have taken particular programming features and properties as the essential starting points, and attempted to coerce a classical logic into a form suitable for a type system, others have

taken the logics themselves as primitive, and attempted to derive new programming disciplines accordingly. In addition, there is a range of work in between, adapting both existing computational and logical presentations in order to find a satisfactory middle-ground.

In this thesis, we are concerned with the question of finding the computational content of classical logic. In particular, we examine classical logics in presentations close to their original definitions, and investigate what natural computational behaviours can be obtained in corresponding term calculi. We subscribe to the increasingly prevalent view that reductions in such term calculi are inherently non-confluent; this is naturally true of cut elimination in classical sequent calculi, as originally defined by Gentzen. Although confluence is a useful practical feature, we believe it is advantageous and instructive to first define an unrestricted, fully-general notion of reductions, and then to examine possible confluent subsystems when the need arises.

We work with Gentzen's two paradigms of sequent calculus and natural deduction. In the case of the sequent calculus, a strong basis of work exists already in defining general canonical reductions. Indeed, a cut elimination procedure dates back to the very first presentation of sequent calculi [39], although for technical reasons it is not general enough to have a good computational interpretation. The question of finding a pleasing generalisation of the cut elimination process which is still strongly normalising was a difficult one (e.g., [31]), but one which we believe has been best tackled in the PhD thesis of Christian Urban [92].

Although the Curry-Howard Correspondence relates *typed* programming calculi with logics, it is interesting in practice to work with *untyped* calculi which come with type assignment systems. However, it is still possible to show that the typed fragment of a calculus has a similar correspondence with a logic, and, so long as all of the reduction rules make sense in the typed case, we believe it is valid to describe even such untyped calculi as having a logical foundation. Conversely, if one takes any set of sound proof transformations (sound in the sense that proofs are always mapped onto proofs), these can always be used to synthesise a typed programming calculus with a Curry-Howard Correspondence. So long as the reduction rules do not depend explicitly on the types, it is then possible to erase the types completely, and obtain an untyped calculus with the same kind of logical foundation. One could imagine, for example, untyped λ-calculus being reinvented from minimal natural deduction in this way.

1.1 The Computational Content of a Logic

It is well known that the reduction relation of the λ-calculus is *confluent*, i.e., that although there are critical pairs in the unrestricted reduction system, these are always joinable. By the Curry-Howard Correspondence, this also implies that the induced notion of reduction on the proofs of implicative minimal logic is confluent. These reductions can be said to come from the work of Prawitz [72], who was interested however only in the *existence* of a normal form for any proof under his reductions, and not in the *uniqueness* of this normal form (which is implied by confluence). From the point of view of *provability*, confluence is an orthogonal concern; whether there are many (normal) proofs or just one for a particular formula is not usually of interest. However, when these proofs are viewed as programs, and the reduction relation defines their semantics, the question of confluence seems an important one since computing different normal forms amounts to a computation producing different answers. This discrepancy can be explained more clearly by the following observation:

The computational content of a logic lies not in its strength in terms of provability, but in its reductions.

Does this mean that the types which are assigned to terms are irrelevant to the computational meaning of these terms? Certainly this is not the case. But the types do not *per se* give us properties about the particular reductions which are possible from a term, and in particular do not guarantee the completeness of those reductions. More generally, the set of *inhabited types* for a particular programming calculus does not directly imply anything about the computational content present in the reductions. We argue that any claim to have extracted the computational content of a logic (by defining a term calculus with a Curry-Howard Correspondence) should not be assessed purely on whether the set of inhabited types coincides with the set of formulas provable in the logic; we view this as a necessary but by no means sufficient requirement.

To take an extreme example to illustrate this point of view, consider the following "logic" in which there is precisely one inference rule, and the judgement *taut* A means "A is a tautology of propositional classical logic".

$$\frac{\textit{taut } A}{\vdash A}\ (\textit{Voilà!})$$

Essentially we have abstracted away *all* of the work in deciding what should be provable, to some other procedure outside of the formalism[1]. It can readily be seen that, if one bases

[1]Technically, this proposal may not actually be considered a propositional logic, if one insists on poly-

an untyped calculus on this "logic", via a Curry-Howard Correspondence, one naturally obtains a syntax with only one object, and an empty reduction relation. The set of inhabited types is exactly the set of formulas valid in classical propositional logic. However, we certainly have not uncovered the computational content of classical logic as a result; it is clear that our calculus has no computational content at all!

The point we wish to make is that the syntactic constructs in the language should be sufficient to faithfully inhabit *all* of the proof steps in the logic. The simplest and most-common case is for each inference rule of the logic to be explicitly represented by a (distinct) syntactic construct in the programming calculus. However, in calculi such as the symmetric λ-calculus [10] and the symmetric $\lambda\mu$-calculus [70], an involutive negation is built in to the definitions, by identifying the types $\neg\neg A$ and A implicitly. In the latter work of Parigot in particular, we believe that this obscures the classical content of the proofs, since (for example) the term $\lambda x.x$ can be typed as a double-negation elimination operator; i.e., assigned the type $\neg\neg A \rightarrow A$. We believe that an explicit treatment of negation is one of the most interesting technical aspects of term calculi based on classical logic, since (as we shall explain in this work) we regard terms of negated type as corresponding to explicit *continuations*, and terms with classical (but not intuitionistic) types as those including computational behaviour associated with *control operators*.

As an implicit consequence of the point of view we are advocating, if one wishes to uncover the computational content of a logic it seems necessary not to restrict the language of proofs in the process (since, by doing so, one is presumably removing reductions from the reduction relation). We will discuss specific examples with regard to this point, but should say here first that we do not think it should be adhered to religiously. In the sequent calculus in particular, one is regularly faced with a number of proofs of the same conclusion (endsequent), which only differ by rather trivial-looking permutations of the inference rules with one another. It seems we would like to work modulo some kind of equivalence relation on these proofs, justified by showing that their computational behaviour is "essentially the same". Some progress has been recently made towards a notion of *proof nets* for classical logic [79, 55], inspired by the solution to the same kinds of problems for linear logic [42]. However, a notion of proof nets has not been found for which cut elimination still corresponds to the original cut elimination in the sequent calculus. Therefore, although it can be argued that the proof nets make a good paradigm to work with in their own right, they are not a direct abstraction of the sequent calculus paradigm. Interestingly, the question of identities on proofs has recently sparked off a whole new paradigm of logics, called *deep inference*, which has been the subject of many recent publications.

nomial time proof-checking [15]. However, it serves to make the point here.

1.2 Canonical Reduction Relations

Given the set of inference rules which make up a logical formalism, the set of proofs in the logic is immediately defined. Therefore, if a programming calculus is to be based on the logic, the syntax (up to choice of notation) and type system (in the case of an untyped calculus) are also implicitly defined. The key ingredient which may still be missing is the reduction rules. Since not all proof systems come with a natural set of reduction rules (for some, a notion of proof normalisation may never have been considered), there remains the problem of deciding what a suitable notion of reduction would be in the resulting programming calculus. The choice of the reduction relation is critical to the question of the computational content: as we have argued above, the computational content of the calculus is essentially defined by the reductions.

In the historical case of the simply-typed λ-calculus and the natural deduction system for intuitionistic implicative logic, the well-known concept of β-reduction seems to be 'the' canonical notion of reduction[2]. Since the λ-calculus was invented before the Curry-Howard Correspondence was observed, there was no need to consider what a suitable set of reductions might be on the underlying logic; these were already specified in the programming calculus. The beauty of the correspondence is that the exact same set of reductions had already been identified as the natural ones for the logic, by Prawitz [72] and (as was recently discovered by von Plato [101]) previously by Gentzen himself.

Prawitz was the first to present a set of proof reductions for classical natural deduction, for the disjunction-free fragment of the logic. In the case of the introduction and elimination inference rules, he follows exactly the same reductions as for intuitionistic logic. The additional reduction rules for classical logic reduce instances of the 'proof by contradiction' rule (called $\curlywedge$C in Prawitz's work) by pushing them outward through the structure of proofs and reducing the degrees of their conclusions until they are restricted to the atomic case. This was sufficient to define a clean characterisation of normal proofs, and to prove normalisation using the reduction rules specified. However, apart from these practical considerations, it is not clear what makes the reduction of the classical rules to the atomic case the 'canonical' notion of reduction.

Since Prawitz's extra reduction rules for classical logic are conditional on the degree of the formulas used in the proofs, it is not possible to use them as the basis of reductions for an untyped term calculus. In contrast, the reduction rules in the intuitionistic case only depend on the *structure* of proofs (in particular, the ordering of inference rules applied), and so can be easily seen to correspond with reduction rules for an untyped calculus.

[2]Sometimes the η rule is also considered to be essential.

Parigot [66] found an adaptation of Prawitz's work which (amongst other results) overcomes this difficulty by restricting and reformulating the classical reduction rules with an aim to obtaining a Curry-Howard Correspondence with a classical logic. This sparked off a wealth of other work, including the discovery of various generalisations of Parigot's original rules. We believe that, unlike in the case of classical sequent calculus, a canonical set of reduction rules for classical natural deduction is not yet settled in the literature, and this is one of the questions we address in this thesis.

1.3 Outline of the Thesis

In Chapter 2, we provide a more detailed background for our work, including further discussions of our goals and point of view. Chapter 3 introduces the term calculus $\mathcal{X}^i$ which we will use as our basis for work on the sequent calculus. It is derived from the work of Christian Urban [92], and its reduction rules reflect his cut elimination procedure. This chapter mainly serves to define the basis for later chapters, but it also contains some new results such as a principal typing algorithm. In Chapter 4, we tackle the notion of *shallow polymorphism* à la ML, in the context of classical logic. This chapter builds on and corrects the work presented in [89]. We show that the naïve generalisation of the historical approach is unsound, in a way which is made particularly clear in the context of the sequent calculus. We define a novel solution to this problem, and prove its soundness.

In Chapter 5 we digress from the realm of sequent calculus and begin work in the natural deduction paradigm. We take Parigot's $\lambda\mu$-calculus as a starting point, and explain why we do not believe it to correspond with a typical system of Gentzen-style natural deduction. We identify specific discrepancies between Parigot's calculus and the system of natural deduction, and by amending these, define a new term calculus which we call $\nu\lambda\mu$. We define a notion of reduction for $\nu\lambda\mu$ which generalises those in the literature, and is motivated by a proposed computational understanding of what the μ-reductions aim to achieve. We show that the resulting notion of reduction is expressive enough to encode Curien and Herbelin's $\overline{\lambda}\mu\tilde{\mu}$-calculus (which is based on classical sequent calculus), which we believe to be a new result for a calculus based strictly on Gentzen-style natural deduction. We also show that a subsyntax of our calculus encodes the λ-calculus, adding additional reduction paths but no new normal forms.

In Chapter 6, we take a step closer to the practical side of functional programming, by comparing the $\nu\lambda\mu$-calculus with *control operators*. The comparison between classical logic and control was the historic catalyst (provided by Griffin [43]) for the great interest which has been shown in Curry-Howard for classical logics since then. We show that our

calculus is able to simulate the reductions of various control operators in the literature, and that, compared with other formulations of $\lambda\mu$, we have more succinct representations for some of the operators. Furthermore, we show that the reductions of $\nu\lambda\mu$ can be seen to contain a 'home-grown' notion of *delimited control*, related to operators such as Felleisen's $\mathcal{F}$ and the *shift*/*reset* operators of Danvy and Filinsky [25]. As a consequence, we put forth the view it is more natural to regard the computational counterpart of double-negation elimination to be a delimited control operator than the $\mathcal{C}$ operator originally studied by Griffin.

In Chapter 7, we work on the relationship between our two calculi, aiming for encodings between the sequent calculus and natural deduction paradigms. We begin with a discussion of existing work towards this goal. We then show how to encode the $\mathcal{X}^i$ calculus into the $\nu\lambda\mu$-calculus, in such a way that reductions and typings are preserved. This provides a strong link between the two calculi. In the other direction, we find that our attempts at encodings are thwarted by the generality of the reductions in $\nu\lambda\mu$. We give some ideas as to how an encoding might be constructed, and explain the difficulties faced.

We conclude in Chapter 8, by discussing how the results of the previous chapters can be brought together to uncover new ones. For example, given the insights provided by Chapter 7, we discuss how the work of Chapter 4 can be adapted to the natural deduction paradigm, providing another novel notion of type-assignment.

Chapter 2

Background Material

2.1 Overview

This chapter provides some of the basic background and history relevant to this thesis. In neither case does it attempt to be complete; its main role is to explain the general framework within which we see our work. In the following chapters, further background and reference material will be introduced and discussed as it is required.

2.2 Notation and Nomenclature

This thesis employs a good deal of mathematical notation, and we wish to make the notation used as clear as possible. Where there is a standard notation in the literature we aim to use this in the thesis, except where it causes conflicts with other notation employed. Where notation is not standardised in the literature we aim to choose one to be as clear as possible.

The following are general points on the notation and terms we use in this document, which we hope may be useful for reference:

1. This document refers to a number of programming calculi. In order to distinguish these (from for example, sequent calculi), we will generically describe these as *term calculi*, and refer to the objects defined by the syntax of a particular calculus as the *terms* of the calculus, except where several distinct classes of syntax are defined, when we will usually follow the nomenclature of the original work.

2. When describing systems of formal logic, we will initially specify the paradigm we are working in (natural deduction, sequent calculus etc.) but later on will often refer to formal proof systems in general simply as *logics*.

3. Since a key aspect of this work is that formulas of the logics and *types* of the term calculi correspond with one another, we will use the same notation for each. This should not cause confusion; rather it illustrates clearly how the Curry-Howard Correspondence (see Section 2.6) operates. We use uppercase Roman characters $A, B, C, \ldots$ to denote formulas and types. These are built from *atomic* formulas/types, for which we use the Greek character φ and $\varphi_1, \varphi_2, \ldots$.

4. Where the syntax of a term calculus involves a single class of variables/names, we will use lowercase Roman characters $x, y, z, \ldots$. We will usually refer to these objects as *term variables* or simply variables. Where a second alphabet of term variables is required, we will use Greek characters $\alpha, \beta, \gamma, \ldots$.

5. We use the terminology 'labeled formula' or 'labeled type' for a term variable paired with a type, written $x : A$. In the context of type systems, this will sometimes be referred to as a *statement*, and can be read as 'x of type A', representing a type assumption for the variable x. In the context of logic, $x : A$ can be viewed as a labeled formula; the x is used to distinguish the occurrence of the formula A from any others which may be available. This allows one to represent the natural multisets of formulas occurring in logical judgements as (isomorphic) sets of labeled formulas.

6. We use uppercase Γ to denote (unordered) sets of formulas/types or (more often) of labeled formulas/types. Whether labels are intended or not is usually clear from the context, but we will be specific when any confusion could arise.

7. We will write judgements, both in logics and in type systems, using the 'turnstile' symbol $\vdash$. These judgements usually feature a set Γ on the left of the turnstile, which represents the set of assumptions under which the judgement is made. In a type system, this amounts to the type assumptions which have been made for the (free) term variables. The notation on the right of the turnstile depends on the particular system in which the judgement features. In a natural deduction system of logic, for example, a judgement may take the form $\Gamma \vdash A$ which denotes that a proof of the formula A has been reached from the assumptions Γ. In a type system, statements may take the form $\Gamma \vdash M : A$ (where M is a term of the syntax), which denotes that M can be assigned type A using the type assumptions in Γ. In some settings, multiple statements occur on the right-hand side of the turnstile. We use uppercase Δ to denote (unordered) sets of formulas/types or (more often) of labeled

formulas/types on the right of a judgement. In this case, judgements may take the form $\Gamma \vdash \Delta$. If a term P is required in judgements of this form, we will usually write it outside, as in $P :\cdot \Gamma \vdash \Delta$. We will sometimes refer to the sets of formulas Γ, Δ collectively as a *context*; specifically the context of the judgement $P :\cdot \Gamma \vdash \Delta$.

8. When particular statements are relevant in a context (e.g., in derivation rules), we use a comma notation: $\Gamma, x : A$ is (for example) a context in which $x : A$ occurs. The comma usually implies a simple set union, but in the case of the premises of derivation rules, we adopt the convention that it represents a *disjoint* union. For example, in the derivation rule:

$$\frac{\Gamma, x : A \vdash \alpha : B, \Delta}{\Gamma \vdash \beta : A{\to}B, \Delta}\,(\to\mathcal{R})$$

the commas in the premise should be read disjointly, implying that $x : A \notin \Gamma$ and $\alpha : B \notin \Delta$, while the comma in the conclusion does not necessarily imply disjointness: the rule can also be applied if $\beta : A{\to}B \in \Delta$.

9. We will present the proofs of all logics we consider in a derivation style, and in a *sequent-style* presentation (i.e., using judgements as described above). There are various other ways to present some of the logics, particularly those of a natural deduction formalism (tree-style, boxproofs etc.), but, for ease of comparison and space considerations, we choose to use sequent derivations as standard.

2.3 Logics

A number of different logics will be discussed during the course of this thesis. We use the term *logic* to indicate a specific language of formulas, along with an identified subset of these formulas which are the *tautologies* of the logic. For example, when we speak of implicative classical logic, we mean a language of formulas containing (only) the implication connective, $\to$, as a constructor, plus (as is always the minimal case) an infinite set of propositional atoms. Furthermore, we intend that the set of formulas such as $A{\to}A$ and $((A{\to}B){\to}A){\to}A$, which are *classically valid*, are the tautologies. However, we do not mean to distinguish different *proof systems* or *reasoning paradigms* as different logics. Instead, we may refer to (for example) implicative classical sequent calculus or implicative classical natural deduction (see later), to specify in which paradigm we are working.

When we wish to speak of a logic in which the syntax of formulas has only one connective, we will usually use adjectives such as "implicative", "conjunctive", etc. However, if we wish to describe a logic having a larger set of connectives, we will usually write which

are included explicitly. For example, "the $\rightarrow$,$\neg$-fragment of classical logic" describes the logic in which formulas are built using only the implication and negation connectives, and the tautologies are exactly those which are classical tautologies. By a "full" logic, we mean a logic in which all of the standard logical connectives are either included in the syntax of formulas or (at least) definable in terms of the connectives which are included. By a fragment, we describe the logic with the same notion of truth, but a syntax of formulas restricted to only some of the standard connectives.

The three most common notions of truth (used to define which are the tautologies of a syntax of formulas) relevant for this thesis define *minimal*, *intuitionistic* and *classical* logics. In a classical logic, all logical connectives are semantically interpreted as specific boolean functions, and a propositional formula is a tautology if and only if it 'evaluates' to true under every assignment of true or false to each distinct propositional atom. In particular, the negation connective $\neg$ is interpreted as the function mapping true to false and false to true, and so, when composed with itself, behaves as the boolean identity function. For this reason, a formula A is a tautology of classical logic if and only if the corresponding formula $\neg\neg A$ is a tautology. In intuitionistic logic, a notion of *constructive argument* characterises which formulas are tautologies. Essentially, a formula A is a tautology if and only if one can provide a 'constructive argument' to deduce A. In particular, it is not acceptable to reason 'by contradiction' in a constructive argument; one must deduce conclusions directly. In an intuitionistic logic, it no longer holds that A is a tautology iff $\neg\neg A$ is; in particular, it does not hold that $\neg\neg A \rightarrow A$ is a tautology of intuitionistic logic. The only situation in which a formula A may (in a sense) be deduced non-constructively is in an *absurd* situation, i.e., when the formula $\bot$ is derivable. In this case, any formula may be derived (for example, by the "$\bot$-elimination" rule in intuitionistic natural deduction). For this reason, $\bot \rightarrow A$ is a tautology of intuitionistic logic. Minimal logic is obtained from intuitionistic logic by restricting even this kind of inference; essentially there is no special meaning (in terms of provability) attached to absurd situations. Therefore, the formula $\bot \rightarrow A$ is not a tautology of minimal logic.

2.4 Gentzen's Formalisms

In [39], the logician Gerhard Gentzen introduced the two most common presentations of formal logic used today, being those of *natural deduction* systems and *sequent calculi*. In this section we will give a brief presentation of the two formalisms, which provide the underlying logical basis for the work discussed in this thesis.

$$\frac{}{\Gamma, A \vdash A}\,(ax) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A{\rightarrow}B}\,(\rightarrow\mathcal{I}) \qquad \frac{\Gamma \vdash A{\rightarrow}B \quad \Gamma \vdash A}{\Gamma \vdash B}\,(\rightarrow\mathcal{E})$$

Figure 2.1: Natural Deduction for Minimal Implicative Logic

2.4.1 Natural Deduction

The intention of the natural deduction formalism is for formal proofs to follow intuitive lines of arguments as much as possible (hence "natural"). Inference rules focus on the *conclusions* which can be drawn from a given set of assumptions. As a natural deduction proof progresses, these assumptions may be *discharged* (certain inference rules may *bind* assumptions) but are usually not added to or manipulated in any other way. In this thesis, assumptions are stored as sets (we typically use the meta-variable Γ to represent these sets).

The inference rules of a natural deduction system mainly come in two distinct varieties: introduction and elimination rules, each for a particular logical connective. Informally speaking, the introduction rules specify when and how a conclusion with the appropriate principal connective can be deduced, whereas the elimination rules specify what can be deduced *from* a conclusion with this principal connective. We write $(\rightarrow\mathcal{I})$ to name the "implication introduction" rule and $(\rightarrow\mathcal{E})$ for "implication elimination" (and use similar rule names in the case of other connectives).

In addition to the introduction and elimination rules, an *axiom* rule is included, which represents trivial deductions of the form "if A is assumed then A can be concluded". Typically this is the only rule with no premise, and so all the 'leaves' of a derivation are instances of the axiom rule.

An example natural deduction calculus is given in Figure 2.1, which shows the standard formulation for minimal (implicative) logic. The rules for the implication connective follow intuitive argument: 'if, by additionally assuming A we can deduce B, then we can deduce $A{\rightarrow}B$' $(\rightarrow\mathcal{I})$ and 'if we know $A{\rightarrow}B$ and we know A then we can deduce B' $(\rightarrow\mathcal{E})$.

Natural deduction is presented by Gentzen in a 'tree style', in which derivations feature single formulas at the nodes, and the formulas at the leaves represent assumptions in the derivation. Thus, one of the obvious differences from sequent calculus systems (to be discussed shortly) was originally in the presentation. However, as has become common practice, we will present both paradigms using sequents. In essence this involves making explicit the assumptions (written on the left of the judgements) which are in scope at each point in a natural deduction derivation.

Natural deduction has remained popular as a means of formalising proofs, and although many aesthetic differences can be seen between existing works in the area, the technical details of the formalism have remained remarkably intact. This can be attributed to the aim of the paradigm to conform with intuitive argument, and the fact that (as a consequence) there seems to be relatively little 'bureaucracy' in the formalism; each inference rule step corresponds with a (sometimes small) step of intuitive argument. However, in terms of meta-theoretical properties, it is somewhat unwieldy, particularly in the case of classical logic. To obtain a natural deduction system for classical logic, one is forced to break the introduction-elimination pattern of the inference rules, and add a rule of "special status". There are many possibilities, as discussed by Gentzen [39]: he opts to include an additional kind of axiom rule, allowing the derivation of a formula $A \vee \neg A$ (for any formula A), commonly known as the 'law of excluded middle'. As an alternative, and one which Gentzen himself later employs for his proofs, one can add one of the rules 'proof by contradiction' or 'double negation elimination':

$$\frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\,(\boldsymbol{PC}) \qquad \frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\,(\neg\neg\mathcal{E})$$

and there exist various other alternatives. However, none of these rules are an introduction or elimination rule for a logical connective, in the traditional sense [39, 72]. Apart from any aesthetic considerations, this makes the definition of a suitable notion of proof reduction problematic (whereas, in the case of minimal logic, a canonical notion of proof reduction is known, due to Gentzen and Prawitz [101, 72]). The definition of a notion of proof reduction can be used to prove meta-theoretical results, such as consistency of the proof system[1]. For example, if it can be shown that all proofs in the system can be reduced to a form in which the *subformula property* holds (which states that all formulas occurring in a proof are subformulas of the conclusion of the proof), then consistency of the formalism can be deduced since it is easy to check that no proof of $\bot$ exists which satisfies the subformula property.

Until recently, it was thought that Gentzen did not even attempt to define proof reduction rules for his natural deduction calculi. However, the discovery by von Plato [101] of a hand-written version of an original manuscript by Gentzen, shows that he worked out a set of proof reduction rules and a proof of normalisation by these rules, for his full intuitionistic natural deduction calculus. On the other hand, in order to prove consistency of his classical natural deduction calculus, he found it necessary to invent his second successful logical paradigm:' the sequent calculus.

[1]Consistency requires that there is no proof of $\bot$ (from no assumptions).

There exists more-recent work by Boričić [19] in the context of classical natural deduction, in which the original Gentzen-style elimination rules have been dropped, and replaced with rules which are more reminiscent of those used in a sequent calculus presentation (see next section) of the logic. The resulting systems evade the difficulties discussed above concerning extra inference rules for negation (which are not required in the setting of the sequent calculus), and it can be argued that they more-accurately reflect the natural semantics of the classical versions of the logical connectives [76]. However, in this thesis we concentrate on natural deduction presentations in their original sense. This is for several reasons. Firstly, the Gentzen-style is still most-commonly used when teaching and writing natural deduction derivations in practice. Secondly, the Gentzen-style presentation of the logic yields a computational representation in the familiar style of the λ-calculus, resulting in clearer connections with functional programming. Finally, from a philosophical point of view, the Gentzen-style presentation of the logic adheres to the original intention of natural deduction, which is to reflect the structure of written mathematical arguments, in a way which we believe alternative reformulations do not.

2.4.2 Sequent Calculus

While natural deduction was designed to try and follow the lines of intuitive argument as much as possible, sequent calculi were introduced for their powerful symmetries, and their usefulness in the proof of technical results. In particular, Gentzen was able to define a set of proof reductions on his sequent calculi known as 'cut elimination' (since they eliminate all instances of the 'cut' rule from any sequent proof). These reductions apply in both the classical and the intuitionistic case, and the cut-free fragments of the sequent calculi are clearly consistent, since they satisfy the subformula property.

Gentzen discovered that with suitably chosen proof rules, he could formulate a very symmetrical system of *classical* sequent calculus by extending sequents to allow more than one formula to occur on the right-hand side. The intended meaning of such a sequent $A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_m$ is 'if all of $A_1, A_2, \ldots, A_n$ are true, then at least one of $B_1, B_2, \ldots, B_m$ is true'. This is rather less intuitive to read than the previous single-conclusion version, but has the advantage of allowing an elegant set of inference rules, which also suffice for intuitionistic logic. Intuitionistic sequent calculus is obtained by Gentzen by simply restricting sequents to only allow *at most* one formula on the right-hand side.

A key feature of sequent calculi is that they do not feature elimination rules for the logical connectives. Instead, all rules are introduction rules, but rules introduce formulas on *either* side of the sequent. The rules for introduction on the right of the sequent (which

we will refer to as *right-introduction rules*) roughly correspond with the introduction rules from natural deduction, and may still be understood as defining when a conclusion featuring the appropriate connective may be deduced. The role of the *left-introduction rules* is slightly less intuitive to understand. However, one may (at least, in the classical case) informally read them as defining the canonical situations in which a formula with a particular principal connective can be shown to be *false*. This can be seen from the fact that any sequent of the form $\Gamma, A \vdash \Delta$ is equivalent (in terms of provability) with the sequent $\Gamma, \neg\Delta \vdash \neg A$, in which $\neg\Delta$ denotes the set of formulas obtained by negating each formula in Δ. We can apply this idea to, for example, the case of implication, in which case the transformation works out as follows:

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A{\rightarrow}B \vdash \Delta}\,(\rightarrow\mathcal{L}) \;\; \textit{becomes} \;\; \frac{\Gamma, \neg\Delta \vdash A \quad \Gamma, \neg\Delta \vdash \neg B}{\Gamma, \neg\Delta \vdash \neg(A{\rightarrow}B)}\,(\rightarrow\mathcal{L}')$$

The translated version of the rule can now be intuitively read as "$A{\rightarrow}B$ is *false* when A is true and B is false", which is indeed the exact criterion for an implication formula to be false in classical logic.

In Gentzen's original presentation of sequent calculi there were a number of *structural* rules, used for manipulating occurrences and positions of formulas in a sequent, without changing the structure of the formulas themselves. Since the work of Kleene [54], it has become popular to make these structural rules implicit in the formulation of the logical inference rules. The key change to make is to deal with *indexed formulas* - in this way contraction can be handled implicitly by automatically merging occurrences of a formula with the same index, while leaving those with different indexes distinct. Similarly, while Gentzen's presentation deals with *ordered* sets of formulas, the addition of labels means that it is straight-forward to identify which formulas are to be bound in an inference rule, and which occurrences remain in the context (without requiring contexts to be ordered). The case of weakening is usually built in to the axiom rule by allowing an arbitrary context in addition to the essential formulas. We follow Kleene in these regards, since in this work, we wish to focus on the logical inferences in a proof, and regard the structural rules as technical considerations we would rather avoid. However, it should be noted that in other work, most notably linear logic, the presence of explicit structural rules is essential[2]. In addition, not all of the implicit techniques described above will work in the case of intuitionistic logic. For example, in the standard presentation of intuitionistic sequent calculus, an explicit weakening rule on the right is still required. For classical logic, which is the focus of this work, we can manage without any explicit structural

[2]This is typically the case for *substructural* logics such as linear logic, in which only some of the classical structural rules are sound.

rules. For an example of a classical sequent calculus in the style of Kleene, see Figure 2.2.

$$\frac{}{\Gamma, x:A \vdash \alpha:A, \Delta}\,(ax) \qquad \frac{\Gamma \vdash \alpha:A, \Delta \quad \Gamma, x:A \vdash \Delta}{\Gamma \vdash \Delta}\,(cut)$$

$$\frac{\Gamma \vdash \alpha:A, \Delta \quad \Gamma, y:B \vdash \Delta}{\Gamma, x:A{\to}B \vdash \Delta}\,(\to\mathcal{L}) \qquad \frac{\Gamma, x:A \vdash \alpha:B, \Delta}{\Gamma \vdash \beta:A{\to}B, \Delta}\,(\to\mathcal{R})$$

$$\frac{\Gamma \vdash \alpha:A, \Delta}{\Gamma, x:\neg A \vdash \Delta}\,(\neg\mathcal{L}) \qquad \frac{\Gamma, x:A \vdash \Delta}{\Gamma \vdash \alpha:\neg A, \Delta}\,(\neg\mathcal{R})$$

Figure 2.2: Classical Sequent Calculus with implication and negation, in the style of Kleene's G3a

The 'cut' rule is the only sequent calculus inference rule (other than the axiom) which is not an introduction rule, and does not (locally) satisfy the subformula property. Gentzen famously proved the "cut elimination theorem" (the *Haupsatz* in [39]), which shows that the cut rule is redundant from the point of view of provability; a proof using the cut rule can always be transformed into a cut-free proof of the same endsequent. The proof defines a constructive procedure for performing such a transformation. Note that Gentzen was only interested in the *existence* of a cut-free proof; in the language of term calculi, Gentzen proved normalisation but not strong normalisation for his set of proof transformations. These transformations are applied to innermost cuts first, and so Gentzen never defines rules for propagating one instance of the cut rule over another. Such permutations are essential for a satisfactory computational interpretation of a cut elimination procedure [92], but also make the question of strong normalisation significantly harder.

The essence of Gentzen's cut elimination procedure can be understood as follows. Consider an arbitrary cut instance:

$$\frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Gamma \vdash \alpha:A, \Delta & \Gamma, x:A \vdash \Delta\end{array}}{\Gamma \vdash \Delta}\,(cut)$$

We will refer to the (labelled) formula occurrences $\alpha:A$ and $x:A$, which are bound in the instance of the cut rule, as the *cut formulas*. The simplest case in the cut elimination is when both cut formulas are the conclusion of the final inference rules of their respective sub-derivations, and furthermore, that no implicit contraction took place (i.e., these occurrences are uniquely introduced in the last step of each sub-derivation). In this case,

we will say the formulas are *introduced* by the sub-derivations. When both formulas are introduced, there is a reduction rule for each possible pair of inference rules introducing the formulas, which removes this instance of the (cut) rule, and constructs a new proof, in which zero or more extra cuts are created of lower *degree*[3]. These reductions will be called *logical rules* or *logical reductions*.

For example, the following reduction is typical of Gentzen-style cut elimination:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma, y:B \vdash \beta:C, \Delta\end{array}}{\Gamma \vdash \alpha:B{\to}C, \Delta}(\to\mathcal{R}) \qquad \dfrac{}{\Gamma, x:B{\to}C \vdash \gamma:B{\to}C, \Delta}(ax)}{\Gamma \vdash \gamma:B{\to}C, \Delta}(cut) \quad \to \quad \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma, y:B \vdash \beta:C, \Delta\end{array}}{\Gamma \vdash \gamma:B{\to}C}(\to\mathcal{R})$$

Logical rules can only applied when both cut formulas are introduced. This can fail to be the case for two reasons: either the formula is not the conclusion of the final inference rule of the sub-derivation, or it is, but the formula also occurs further up in the derivation, and an implicit contraction takes place here. In either case, the cut can be dealt with by *propagating* it through the structure of the appropriate sub-derivation, seeking out the occurrences of the cut formula. A copy of the cut is deposited next to each such occurrence. For example, the following reduction can be made:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_3\\ \Gamma, x:A, y:A \vdash \beta:B, \alpha:B, \Delta\end{array}}{\Gamma, x:A \vdash \gamma:A{\to}B, \alpha:B, \Delta}(\to\mathcal{R})}{\Gamma \vdash \gamma:A{\to}B, \Delta}(\to\mathcal{R}) \qquad \begin{array}{c}\mathcal{D}_4\\ \Gamma, z:A{\to}B \vdash \Delta\end{array}}{\Gamma \vdash \Delta}(cut)$$

$$\to$$

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_3\\ \Gamma, x:A, y:A \vdash \beta:B, \alpha:B, \Delta\end{array}}{\Gamma, x:A \vdash \gamma:A{\to}B, \alpha:B, \Delta}(\to\mathcal{R}) \qquad \begin{array}{c}\mathcal{D}_4\\ \Gamma, x:A, z:A{\to}B \vdash \alpha:B, \Delta\end{array}}{\Gamma, x:A \vdash \alpha:B, \Delta}(cut)}{\Gamma \vdash \epsilon:A{\to}B, \Delta}(\to\mathcal{R}) \qquad \begin{array}{c}\mathcal{D}_4\\ \Gamma, z:A{\to}B \vdash \Delta\end{array}}{\Gamma \vdash \Delta}(cut)$$

Note that we have renamed one occurrence of γ to a fresh name ϵ during the propagation

[3]The degree of a cut is the degree of its cut formula A, which (as is standard) is the maximum of the depths of the branches of its syntax tree.

of the cut. This is not strictly necessary, but emphasises the fact that these two formula occurrences are now bound in different cuts. Although this example illustrates *left* propagation of a cut, the analogous reductions may take place when the formula bound on the right of a cut is not introduced. In the special case of a cut formula not being the conclusion of *any* inference rule in the sub-derivation (i.e., it was only present in the sequent due to a weakening of the context), the act of propagating the cut causes the entire cut and other sub-derivation to be erased. In particular, when *both* cut formulas are of this kind, a cut can be reduced to either one of its sub-derivations, non-deterministically. This illustration of the non-determinism inherent in the cut elimination process has become known as "Lafont's example", and was discussed in [40]. The status of this inherent non-confluence is discussed further in Section 2.10.

2.5 Prawitz's Proof Reductions

In his most famous work [72], Prawitz made an extensive study of Gentzen's natural deduction calculi, and made various important contributions to the area. He defined the *inversion principle*, which is a proposal to formalise criteria to characterise *good* introduction and elimination rules for a logical connective. Essentially, this states that no more should be derivable by the elimination rule(s) than is contained within the premises of the introduction rules. Implicitly, this suggests that whenever an elimination rule is applied whose main formula is the consequence of an introduction rule, the conclusion derived was already in some sense available earlier in the proof. This leads to the basis of the set of proof reduction rules which Prawitz defines, firstly for intuitionistic natural deduction, and then extends to classical natural deduction. For both systems, Prawitz defines a suitable notion of *normal proof* (one which cannot be further reduced by his proof reductions), and is able to show that all proofs can be reduced to such a normal form. From these results, he is able to directly establish the consistency of the natural deduction calculi, in a similar fashion to Gentzen's arguments regarding cut elimination in the sequent calculus.

Prawitz [72] made an extensive study of normalisation in natural deduction systems for minimal, intuitionistic and classical logics. In the case of classical logic, he was the first to define a set of reduction rules to deal with the propagation of occurrences of "proof by contradiction" (PC) within a proof[4].

The aim of Prawitz's reduction rules concerning the (PC) rule is to provide proof trans-

[4]In his work, the "proof by contradiction" rule is called ⅄C: he uses the symbol ⅄ instead of $\bot$, and the C denotes "classical".

formations which reduce the 'degree' of the formula concluded by the (PC) rule. To this end, he defined a reduction rule for each logical connective, each of which reduced the degree of the concluded formula by one. For example, in the case of implication, the appropriate rule is:

$$\dfrac{\begin{array}{c}\mathcal{D}\\ \Gamma, x:\neg(A{\rightarrow}B)\vdash\bot\end{array}}{\Gamma\vdash A{\rightarrow}B}\,(PC) \quad\rightarrow\quad \dfrac{\dfrac{\begin{array}{c}\mathcal{D}^*\\ \Gamma, y:A, w:\neg B\vdash\bot\end{array}}{\Gamma, y:A\vdash B}\,(PC)}{\Gamma\vdash A{\rightarrow}B}\,(\rightarrow\mathcal{I})$$

in which $\mathcal{D}^*$ denotes the derivation obtained by taking $\mathcal{D}$ and replacing all axioms introducing the statement $x:\neg(A{\rightarrow}B)$ with the following derivation (in which some occurrences of assumptions in contexts have been deleted to save space):

$$\dfrac{\dfrac{\dfrac{}{\Gamma, w:\neg B\vdash\neg B}\,(ax) \qquad \dfrac{\dfrac{}{\Gamma, z:A{\rightarrow}B\vdash A{\rightarrow}B}\,(ax) \qquad \dfrac{}{\Gamma, y:A\vdash A}\,(ax)}{\Gamma, y:A, z:A{\rightarrow}B\vdash B}\,(\rightarrow\mathcal{E})}{\Gamma, y:A, w:\neg B, z:A{\rightarrow}B\vdash\bot}\,(\neg\mathcal{E})}{\Gamma, y:A, w:\neg B\vdash\neg(A{\rightarrow}B)}\,(\neg\mathcal{I})$$

Prawitz defines a similar rule for each logical connective in his formulation of natural deduction. However, because applicability of these rules is conditional on the particular formula concluded by the (PC) rule, they cannot be adapted into an untyped version suitable for incorporation into an untyped calculus. If one were to discard the conditions, and for example to blindly apply the rule above in an untyped setting, it would obviously violate any kind of strong normalisation result, since any μ-bound term would always be a redex, and would always reduce to a term containing a μ-binder. However, these concerns are irrelevant to the work of Prawitz, since he was considering only transformations of *proofs*, in which (obviously) formulas occur explicitly. Using these rules, he was able to show that it is always possible to reduce a proof to a normal form in which (amongst other restrictions) occurrences of the (PC) rule are limited to atomic conclusions. This choice aids the proof of normalisability. From this result, Prawitz was able to deduce consistency of the natural deduction calculus directly, just as Gentzen had done for cut-free sequent calculus.

Until recently [101], Prawitz's work was thought to be the first to consider a full system of proof normalisation for natural deduction calculi. It is still considered to be the seminal

work on the meta-theory of natural deduction calculi, and was the first to attempt to give a concrete criterion for which sets of introduction and elimination rules are acceptable in such a calculus, as well as the first to define such proof reductions for *classical* natural deduction.

Prawitz's proof reductions form the basis of later work by Parigot, as we will describe in Chapter 5. Prawitz also gives, towards the end of his work, a brief account of sequent calculi, and gives a verbal description of an encoding of natural deduction proofs into sequent calculi. His encoding is an improvement on Gentzen's, since (in at least the intuitionistic case) he maps normal terms onto cut-free proofs. Encodings between these two disciplines will be discussed in more detail in Chapter 7.

2.6 The Curry-Howard Correspondence

The Curry-Howard Correspondence describes a one-to-one correspondence between the terms of the simply-typed lambda calculus [20][5] and the natural deduction proofs in a standard system for minimal implicative logic. However, the correspondence (which is sometimes called the Curry-Howard *Isomorphism*) is deeper than this: the types of λ-calculus terms become formulas in the logic, while reduction (in the usual sense for the λ-calculus) corresponds precisely with *proof normalisation* in the logic. The correspondence was the result firstly of observations by Curry [22, 23] about the relationship between his *combinatory logic* and a Hilbert-style formalisation of minimal logic, and secondly by Howard [50] concerning the relationship between λ-calculus and minimal natural deduction. At approximately the same time as Howard's observation, de Bruijn was using a λ-calculus notation to describe proofs, and for this reason, the correspondence is sometimes referred to as the Curry-Howard-de Bruijn Correspondence. Essentially for brevity, we will refer to the Curry-Howard Correspondence in this work.

Although the correspondence was originally made between *typed* λ-calculus and natural deduction, it is perhaps easiest to see the striking similarities when one considers the simple type-assignment system for the *untyped* λ-calculus:

[5] We assume the reader to be familiar with the λ-calculus.

Definition 2.6.1 (Illustration of the Curry-Howard Correspondence)**.**

λ-calculus	*Minimal Natural Deduction*
$\dfrac{}{\Gamma, x:A \vdash_\lambda x:A}\,(ax)$	$\dfrac{}{\Gamma, x:A \vdash A}\,(ax)$
$\dfrac{\Gamma, x:A \vdash_\lambda M:B}{\Gamma \vdash_\lambda \lambda x.M : A{\to}B}\,(\to\mathcal{I})$	$\dfrac{\Gamma, x:A \vdash B}{\Gamma \vdash A{\to}B}\,(\to\mathcal{I})$
$\dfrac{\Gamma \vdash_\lambda M:A{\to}B \quad \Gamma \vdash_\lambda N:A}{\Gamma \vdash_\lambda M\ N:B}\,(\to\mathcal{E})$	$\dfrac{\Gamma \vdash A{\to}B \quad \Gamma \vdash A}{\Gamma \vdash B}\,(\to\mathcal{E})$

It is clear that each inference rule from the logic corresponds to the way in which types are derived for a syntax construct in the λ-calculus. Furthermore, the correspondence covers reduction: the familiar (β) reduction rule of the λ-calculus corresponds to the following proof reduction:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma, x:A \vdash B\end{array}}{\Gamma \vdash A{\to}B}\,(\to\mathcal{I}) \qquad \begin{array}{c}\mathcal{D}_2\\ \Gamma \vdash A\end{array}}{\Gamma \vdash B}\,(\to\mathcal{E}) \quad \to \quad \begin{array}{c}\mathcal{D}_1\langle\mathcal{D}_2/x\rangle\\ \Gamma \vdash B\end{array}$$

in which $\mathcal{D}_1\langle\mathcal{D}_2/x\rangle$ denotes the derivation obtained by starting from $\mathcal{D}_1$ and replacing every instance of the (ax) rule of the form $\Gamma, x:A \vdash A$ is replaced by a copy of the derivation $\mathcal{D}_2$, concluding $\Gamma \vdash A$.

2.6.1 Two Views of the Extension of Curry-Howard

This thesis concerns the extension of the Curry-Howard Correspondence to the setting of classical logic. There are two opposite viewpoints which can be identified concerning this question. Firstly, one could consider existing programming calculi which seem appropriate, and examine to how to adapt them in order to make a type system based on a suitably chosen (and possibly modified) formulation of classical logic feasible. In doing so, one might make changes, restrictions or extensions to the original programming calculus, and the logical formalism. This way of working was the original one, since it is essentially what Griffin achieves in his seminal paper [43]. He takes an existing calculus and shows how, given certain restrictions and modifications to the syntax, one can give a type system corresponding to classical natural deduction.

The alternative point of view, which seems to be more recent, is to take a formulation of classical logic as the starting point. If this logic is equipped with a suitable notion of proof reductions, then it is straight-forward to choose an appropriate term representation for the proofs of the logic, and to define corresponding reduction rules on this syntax. Furthermore, by taking the untyped version of such a calculus, one can immediately obtain an untyped calculus based on this formulation of classical logic. This approach seems to us to be neatly summarised by the question, "what is the computational content of classical logic?". The advantage of the approach is that this question can in principle be convincingly answered, so long as one remains faithful to the original logic. The disadvantages are two-fold. Firstly, if a suitable notion of proof reduction is not already known for the particular formulation of classical logic, then it must be defined before a suitable term calculus can be synthesised. Furthermore, one must argue for the suitability of the chosen notion of proof reduction. Secondly, having stayed faithful to the logic, it is clear that a clean Curry-Howard style correspondence may be achieved, but it is less clear what the actual computational meaning of the resulting term syntax is.

In this work, we adhere strongly to the second viewpoint described above. We are interested in investigating exactly what computational content can be discovered if one attempts to build term calculi based on canonical formulations of classical logic. In particular, we focus on Gentzen's systems of classical natural deduction and classical sequent calculus, although in the latter case we adopt the structural-rule-free style which has become popular since the work of Kleene [54].

Perhaps the best examples of work which take the logic as the starting point, are the works of Urban [92] (see Section 2.8) and Lengrand [56], which led to the work of van Bakel et al. [98] (see Section 2.9). The work of Curien and Herbelin [21] also has close ties with Gentzen's presentation of classical logic, although it does differ in some respects (see Section 5.5.4). It is interesting to note that all of these works are based on classical *sequent calculus* rather than natural deduction. In the natural deduction paradigm, relatively little work exists concerning a direct correspondence with Gentzen's classical natural deduction. Perhaps because of the similarities with the paradigm of the λ-calculus, attempts have mostly been made to coerce presentations of classical natural deduction in such a way that certain properties and syntactic constructs historically identified with the study of λ-calculi are preserved. In particular, much of the work, arguably including the most famous of Parigot [66], focuses on constructing calculi which are *confluent*. However, as we shall argue in Section 2.10 (and has been argued previously by, e.g., Urban [92]), this is not a natural property for a general set of reductions for classical logic to possess, and we believe that the assumption *a priori* that such a property is an essential requirement restricts the study and understanding of the full computational content of classical logic.

2.6.2 Curry-Howard for Untyped Calculi?

The original Curry-Howard correspondence was made using *typed* λ-calculus. It is not possible to form precisely the same kind of correspondence with *untyped* λ-calculus for two (related) reasons: firstly, there are untypeable terms in the syntax of the calculus, which do not correspond to proofs, and secondly, the typeable terms can each be assigned types in an infinite number of ways. This latter objection does not seem too serious, since each typeable term has a *principal typing*, and one could consider using these as the basis for a correspondence. However, the former point is more serious. On the other hand, from a programming perspective, it is generally a requirement to be able to express recursive functions, which are not naturally typeable in a simple type system. This can be managed by artificially adding a notion of typeable recursion external to the logic underlying the calculus, as in the case of ML [49, 58], or by extending the type-system to a theory which can satisfy recursive type constraints. However, the approach which we work with in this thesis is to study the untyped versions of the calculi, since this still allows the type system (and syntax) to closely maintain its correspondence with propositional logic. Therefore, we wish to generalise the idea of the Curry-Howard Correspondence to untyped calculi.

If one takes any typed programming calculus, it can be used to define the syntax of an untyped programming calculus by erasing all type information from the syntax. Furthermore, the structure of the original type-information usually immediately implies a suitable type system for the untyped syntax. It is clear that the syntax and type system of untyped λ-calculus can be obtained from the definitions of typed λ-calculus in this way. Note that the resulting syntax may be more-general: for example, the untyped λ-calculus term $x\ x$ does not have an analogue in the standard typed version. As far as reduction rules are concerned, there is a subtle point to consider. In general, one can easily adapt the reduction rules of the typed calculus to an untyped version by erasing type-information in the terms. However, in some cases it may be that typed calculi have reductions defined in which explicit conditions are made on the types of terms. For example, in the typed λ-calculus extended with *pairing* (or conjunction, in the type language), one might express the rule for (η)-expansion as $M \rightarrow \lambda x.(M\ x)$ *if* $M : A \rightarrow B$. The condition on the types ensures the soundness of the rule, since, if M is instead a pair, the resulting term would be invalid were the rule applied. However, it is not clear how to define an untyped analogue of this reduction rule, since the explicit condition on the types is essential for soundness. Such rules are relatively rare in the literature, and, in particular, most of the reduction rules which will be considered in this thesis will not be of this problematic kind. Therefore, we content ourselves with the fact that the definitions of typed calculi may yield definitions of corresponding untyped calculi *provided* such explicit type-conditions on the reduction rules do not exist.

Conversely, it is straightforward to see that the definitions of an untyped calculus equipped with a sound type system (satisfying subject reduction) can be used to define a corresponding typed calculus by considering only the sub-syntax of typeable terms, modifying the syntax of such terms to carry appropriate type information explicitly, and applying the corresponding reduction rules to these terms. This is often referred to as the *typed* restriction of a calculus. The only condition that should be added is that all of the original reduction rules have at least some typeable instances; i.e., none are lost in the typed restriction.

We can now define what we mean by an untyped calculus based on an untyped correspondence:

Definition 2.6.2 (Curry-Howard for untyped calculi)**.** *We say that an untyped calculus* Π *(with a sound type system), is based on a Curry-Howard correspondence with a particular formulation of logic, if the typed restriction of* Π *has a Curry-Howard correspondence with the logic, in the sense of the original correspondence.*

2.7 Griffin's Observation

The $\lambda\mathcal{C}$-calculus was the first programming calculus for which a connection with classical logic was made. The calculus itself was invented by Felleisen et. al. [37], in an attempt to formalise programming constructs which were already in use in languages at the time. The calculus is essentially a λ-calculus extended with two unary operator on terms, $\mathcal{C}$ (pronounced, "control") and $\mathcal{A}$ (pronounced, "abort"), which provide the facility to manipulate the *context* in which an execution takes place. These operators are examples of *control operators*. The $\lambda\mathcal{C}$-calculus itself is defined as follows:

Definition 2.7.1 (The $\lambda\mathcal{C}$-Calculus [37, 43][6])**.** *Terms* M, N*, applicative contexts* C *and reduction* $\rightarrow$ *are defined as follows, for* $\lambda\mathcal{C}$*, in which* $C\{M\}$ *denotes the insertion of the term* M *into the 'hole'* $\bullet$ *of* C*, and in which values* V *are defined to be variables or terms of the form* $\lambda x.M$*, as usual:*

$$
\begin{array}{lll}
M, N ::= x & & \\
\quad \mid \lambda x.M & C ::= \bullet & C\{(\lambda x.M)\ V\} \rightarrow C\{M\langle V/x\rangle\} \\
\quad \mid M\ N & \quad \mid C\ M & C\{\mathcal{A}(M)\} \rightarrow M \\
\quad \mid \mathcal{C}(M) & \quad \mid V\ C & C\{\mathcal{C}(M)\} \rightarrow M\ (\lambda z.\mathcal{A}(C\{z\})) \\
\quad \mid \mathcal{A}(M) & &
\end{array}
$$

[6] Originally, this calculus was referred to as 'Idealised Scheme', since it was thought to be modelling the call/cc construct from Scheme. However, the name $\lambda\mathcal{C}$ has become common since the work of Griffin [43].

The operator $\mathcal{A}$ provides the simplest control over its surrounding context, by completely discarding it. The operator $\mathcal{C}$, on the other hand, removes the current context, but forms a λ-abstracted version of the context surrounded by an $\mathcal{A}$ operator, and passes this λ-abstraction to the argument M. The effect is that M has flexible control over the context; during execution, if a value is passed to the term $\lambda z.\mathcal{A}(C\{z\})$, then the $\mathcal{A}$ will cause other execution to be terminated, and the original context will be reinstated with the chosen value in position. On the other hand, if no value is ever passed to the abstraction, M is free to evaluate normally, and even to discard the abstracted copy of the context. In particular, it is possible to treat $\mathcal{A}$ as a defined construct: $\mathcal{A}_{\mathcal{C}}(M) =_{def} \mathcal{C}(\lambda z.M)$ *where* $z \notin fv(M)$.

Griffin [43] considered the question of typing these control operators, and observed that it is possible to assign the logically-consistent types $\bot\rightarrow A$ to $\mathcal{A}$ and $((A\rightarrow\bot)\rightarrow\bot)\rightarrow A)$ to $\mathcal{C}$. Thus, it appears that $\mathcal{A}$ could be viewed as the computational counterpart of the natural deduction $(\bot\mathcal{E})$ rule, while $\mathcal{C}$ could be viewed as that of double-negation elimination. This was the first time that an extension of the Curry-Howard correspondence to a classical logic was considered, and sparked a whole new research field.

While the significance of this observation should not be underrated, it is less clear now that $\mathcal{C}$ actually *is* the computational counterpart of double-negation elimination. Firstly, as Griffin himself observed, the reduction rules above are not fully consistent with the typings proposed, in the sense that subject reduction is not guaranteed. In particular, a term of the form $C\{\mathcal{A}(M)\}$ may in principle have any type, while the subterm M is required to have type $\bot$. Therefore, by applying the reduction rule associated with $\mathcal{A}$, we observe a subject reduction problem. Griffin proposed a workaround for this issue, by essentially 'wrapping' all programs in a special context which was guaranteed to have type $\bot$. However, this kind of restriction is somewhat arbitrary from the point of view of the *logic*. Secondly, the most general logically-consistent typing for $\mathcal{C}$ is not $((A\rightarrow\bot)\rightarrow\bot)\rightarrow A$, but rather $((A\rightarrow B)\rightarrow\bot)\rightarrow A$, for any types A and B. The cause for this discrepancy is the occurrence of the $\mathcal{A}$ operator in the reduction rule for $\mathcal{C}$. If it were removed, then the most general logically-consistent typing for $\mathcal{C}$ would indeed become double-negation elimination. Furthermore, Felleisen later introduced an operator $\mathcal{F}$ [38] whose reduction semantics do just this; the $\mathcal{A}$ is not present. Since it is the occurrence of $\mathcal{A}$ in the reduction rule which causes the subject reduction problems, the $\mathcal{F}$ operator has no such issues. We will discuss these matters further in Chapter 6, where we give further arguments to support the point of view that $\mathcal{C}$ is not the canonical inhabitant of double-negation elimination[7].

[7] It is only fair to point out that Griffin did not claim this fact himself; his contribution was not attempting to present the 'best' computational interpretation of classical logic, but rather to observe that there could be a relationship at all!

2.8 Urban's Cut Elimination

In his PhD thesis [92], Urban presents a new set of cut elimination rules for the classical sequent calculus. His work is motivated by a desire to obtain a notion of cut elimination which is as general as possible, while still obtaining a strong normalisation result for his set of reduction rules. By "generality", it is meant that the natural non-confluence of classical cut-elimination, rather being regarded as a drawback, should be kept unrestricted as far as possible. In particular, it is hoped that as many different normal proofs as possible can be reached by reduction of a proof.

Urban's cut elimination is based on the very general notion of reduction which is obtained by adding to Gentzen's original set of reduction rules (used in the proof of his *Haupsatz*) rules for allowing one cut to 'cross-over' another one. As Urban explains, these kinds of reductions are necessary in order to have a chance of a good computational interpretation of the reductions: without them, one cannot even simulate the reductions of the λ-calculus (since, essentially, these rules allow the evaluation of a substitution to pass through further redexes which have not yet been evaluated). However, the unrestricted use of these rules causes strong normalisation to immediately fail (in the simplest case, two cuts can be made to cross-over one another indefinitely). Urban provides a solution to this problem, related to the previous work of Barbanera et al. [10, 11, 12], and produces two cut elimination procedures; one of which treats the propagation of cuts as an immediate implicit operation (similarly to the status of substitution in the λ-calculus), and one which represents propagation syntactically as a step-by-step process (similarly to the *explicit substitutions* found in λx [18]). The latter case is presented because it provides a cut elimination process based only on *local* rewriting rules, which is the style in which Gentzen originally defines cut elimination. This latter cut elimination procedure was later incorporated into the work of Lengrand [56], and forms the basis of the work of van Bakel et. al. [98] (see Section 2.9), whose origins are an untyped version of Urban's term annotations. On the other hand, in this thesis we will study the former version of Urban's work, since our contributions are orthogonal to the exact status of propagation in the cut elimination, and we believe the presentation is slightly simpler. We will, however, employ an adaptation of the syntax of [98], since we find this easier to work with and explain than the prefix notation of Urban's work.

2.9 The $\mathcal{X}$-Calculus

The $\mathcal{X}$-calculus is based on an untyped term annotation related to classical implicative sequent calculus, derived from the work of Urban [92] and Lengrand [56]. In this section

we recall the basic definitions (which come from the work of van Bakel et al. [98]).

Definition 2.9.1 ($\mathcal{X}$-Terms)**.** *The terms of the $\mathcal{X}$-calculus are defined by the following syntax, where x, y range over the infinite set of* sockets *and α, β over the infinite set of* plugs *(sockets and plugs together form the set of* connectors*).*

$$\begin{array}{rcccccc} P, Q ::= & \langle x.\alpha \rangle & \mid \widehat{y}P\widehat{\beta}\cdot\alpha & \mid P\widehat{\beta}\,[y]\,\widehat{x}Q & \mid P\widehat{\alpha}\dagger\widehat{x}Q & \mid P\widehat{\alpha} \not\prec \widehat{x}Q & \mid P\widehat{\alpha} \not\succ \widehat{x}Q \\ & \text{capsule} & \text{export} & \text{import} & \text{cut} & \text{left-cut} & \text{right-cut} \end{array}$$

The $\widehat{\cdot}$ symbolises that the connector underneath is bound in the attached subterm—a bound socket is written as a prefix to the term, whereas a bound plug is written as a suffix. For example in the import $P\widehat{\beta}\,[y]\,\widehat{x}Q$, occurrences of β are bound in the subterm P and occurrences of x are bound in Q. A connector which does not occur under a binder is said to be free. We will use *fp*(P) to denote the free plugs of P, and similarly *fs*(P) for free sockets. The reduction rules are specified below.

Definition 2.9.2 (Logical Rules)**.** *The logical rules are presented by:*

$$\begin{array}{lrll} (\textit{cap}): & \langle y.\alpha\rangle\widehat{\alpha}\dagger\widehat{x}\langle x.\beta\rangle & \to \langle y.\beta\rangle & \\ (\textit{impR}): & (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}\langle x.\gamma\rangle & \to \widehat{y}P\widehat{\beta}\cdot\gamma & \alpha \notin \textit{fp}(P) \\ (\textit{impL}): & \langle y.\alpha\rangle\widehat{\alpha}\dagger\widehat{x}(P\widehat{\beta}\,[x]\,\widehat{z}Q) & \to P\widehat{\beta}\,[y]\,\widehat{z}Q & x \notin \textit{fs}(P,Q) \\ (\textit{imp}): & (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R) & \to \left\{\begin{array}{l} Q\widehat{\gamma}\dagger\widehat{y}(P\widehat{\beta}\dagger\widehat{z}R) \\ (Q\widehat{\gamma}\dagger\widehat{y}P)\widehat{\beta}\dagger\widehat{z}R \end{array}\right\} & \begin{array}{l}\alpha \notin \textit{fp}(P), \\ x \notin \textit{fs}(Q,R)\end{array} \end{array}$$

The first three logical rules above specify a renaming (reconnecting) procedure, whereas the last rule specifies the basic computational step: it allows the body of the function from the export to be inserted between the two subterms of the import (the resulting cuts may be bracketed either way, as shown). In logical terms, this reduction corresponds to the logical cut elimination rule for implication.

Definition 2.9.3 (Activation Rules)**.** *We define two* cut-activation *rules.*

$$\begin{array}{l} (\textit{act}\text{-L}): P\widehat{\alpha}\dagger\widehat{x}Q \to P\widehat{\alpha} \not\prec \widehat{x}Q \textit{ if } P \textit{ does not introduce } \alpha \\ (\textit{act}\text{-R}): P\widehat{\alpha}\dagger\widehat{x}Q \to P\widehat{\alpha} \not\succ \widehat{x}Q \textit{ if } Q \textit{ does not introduce } x \end{array}$$

where: P introduces x: Either $P = Q\widehat{\beta}\,[x]\,\widehat{y}R$ and $x \notin \textit{fs}(Q,R)$, or $P = \langle x.\alpha\rangle$

P introduces α: Either $P = \widehat{x}Q\widehat{\beta}\cdot\alpha$ and $\alpha \notin \textit{fp}(Q)$, or $P = \langle x.\alpha\rangle$

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. Whenever an active cut meets a circuit exhibiting the connector it is trying to communicate with, a new (inactive)

cut is 'deposited', representing an attempt to communicate at this level. The pushing of the active cut continues until the level of capsules is reached, where it is either deactivated or destroyed. Once again, the inactive cut can reduce via a logical rule, or pushing can continue in the other direction. This behaviour is expressed by the following propagation rules.

Definition 2.9.4 (Propagation Rules)**.** *Left Propagation:*

$$\begin{aligned}
(\not\dagger): &\quad \langle y.\alpha\rangle\widehat{\alpha} \not\leftarrow \widehat{x}P \to \langle y.\alpha\rangle\widehat{\alpha} \dagger \widehat{x}P \\
(\mathit{cap}\not): &\quad \langle y.\beta\rangle\widehat{\alpha} \not\leftarrow \widehat{x}P \to \langle y.\beta\rangle && \beta \neq \alpha \\
(\mathit{exp\text{-}outs}\not): &\quad (\widehat{y}Q\widehat{\beta}\cdot\alpha)\widehat{\alpha} \not\leftarrow \widehat{x}P \to (\widehat{y}(Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{x}P, && \gamma \mathit{\ fresh} \\
(\mathit{exp\text{-}ins}\not): &\quad (\widehat{y}Q\widehat{\beta}\cdot\gamma)\widehat{\alpha} \not\leftarrow \widehat{x}P \to \widehat{y}(Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta}\cdot\gamma, && \gamma \neq \alpha \\
(\mathit{imp}\not): &\quad (Q\widehat{\beta}\,[z]\,\widehat{y}R)\widehat{\alpha} \not\leftarrow \widehat{x}P \to (Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta}\,[z]\,\widehat{y}(R\widehat{\alpha} \not\leftarrow \widehat{x}P) \\
(\mathit{cut}\not): &\quad (Q\widehat{\beta} \dagger \widehat{y}R)\widehat{\alpha} \not\leftarrow \widehat{x}P \to (Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta} \dagger \widehat{y}(R\widehat{\alpha} \not\leftarrow \widehat{x}P)
\end{aligned}$$

Right Propagation:

$$\begin{aligned}
(\not\backslash\dagger): &\ P\widehat{\alpha} \not\backslash \widehat{x}\langle x.\beta\rangle && \to P\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle \\
(\not\backslash\mathit{cap}): &\ P\widehat{\alpha} \not\backslash \widehat{x}\langle y.\beta\rangle && \to \langle y.\beta\rangle, && y \neq x \\
(\not\backslash\mathit{exp}): &\ P\widehat{\alpha} \not\backslash \widehat{x}(\widehat{y}Q\widehat{\beta}\cdot\gamma) && \to \widehat{y}(P\widehat{\alpha} \not\backslash \widehat{x}Q)\widehat{\beta}\cdot\gamma \\
(\not\backslash\mathit{imp\text{-}outs}): &\ P\widehat{\alpha} \not\backslash \widehat{x}(Q\widehat{\beta}\,[x]\,\widehat{y}R) && \to P\widehat{\alpha} \dagger \widehat{z}((P\widehat{\alpha} \not\backslash \widehat{x}Q)\widehat{\beta}\,[z]\,\widehat{y}(P\widehat{\alpha} \not\backslash \widehat{x}R)), \\
& && && z \mathit{\ fresh} \\
(\not\backslash\mathit{imp\text{-}ins}): &\ P\widehat{\alpha} \not\backslash \widehat{x}(Q\widehat{\beta}\,[z]\,\widehat{y}R) && \to (P\widehat{\alpha} \not\backslash \widehat{x}Q)\widehat{\beta}\,[z]\,\widehat{y}(P\widehat{\alpha} \not\backslash \widehat{x}R), && z \neq x \\
(\not\backslash\mathit{cut}): &\ P\widehat{\alpha} \not\backslash \widehat{x}(Q\widehat{\beta} \dagger \widehat{y}R) && \to (P\widehat{\alpha} \not\backslash \widehat{x}Q)\widehat{\beta} \dagger \widehat{y}(P\widehat{\alpha} \not\backslash \widehat{x}R)
\end{aligned}$$

We write $\to$ for the reduction relation generated by the logical, propagation and activation rules. The following are admissible rules (see [98, 99]).

Lemma 2.9.5 (Garbage Collection and Renaming)**.**

$$\begin{aligned}
&(\not\mathit{gc}): P\widehat{\alpha} \not\leftarrow \widehat{x}Q \to P, \text{ if } \alpha \notin \mathit{fp}(P) && (\mathit{ren}\text{-L}): P\widehat{\delta} \dagger \widehat{z}\langle z.\alpha\rangle \to P[\alpha/\delta] \\
&(\not\backslash\mathit{gc}): P\widehat{\alpha} \not\backslash \widehat{x}Q \to Q, \text{ if } x \notin \mathit{fs}(Q) && (\mathit{ren}\text{-R}): \langle z.\alpha\rangle\widehat{\alpha} \dagger \widehat{x}P \to P[z/x]
\end{aligned}$$

2.10 The Question of Confluence

One question on which the existing literature on Curry-Howard for classical logic seems not to have reached a consensus is that of *confluence*. It is a well-known result that the reductions of the λ-calculus are confluent, meaning that although non-deterministic choices can be made in (unrestricted) reduction, it is always the case that the alternatives

have a common reduct. In particular, this guarantees that whenever a λ-term has a normal form, the normal form is unique. This can be used as the basis of a semantics for the calculus, and in general makes reasoning about meta-theoretical results much simpler.

Because of its practical advantages, and natural occurrence in the field of λ-calculi, confluence has been regarded as an essential result of almost all functional languages proposed since. Where non-confluence is discovered, it is usual for the definitions to be altered, and reductions restricted where necessary in order to guarantee the result. This attitude can be justified by the fact that, in the underlying computational model (λ-calculus), confluence is a natural and essential property. Since the study of a Curry-Howard Correspondence with classical logic had its origins in the study of existing functional calculi, it is only natural that the initial attitude was that confluence remained an expected and necessary feature of calculi to be studied.

However, when one returns to the origins of reductions for classical logic, it is clear that confluence is *not* a naturally-occurring phenomenon. In particular, Gentzen's definition of cut elimination for classical sequent calculus is inherently non-confluent. We believe it is important to separate the notion of what the Curry-Howard Correspondence means fundamentally from the properties commonly associated with it. In particular, while it is of course the case that the original Curry-Howard Correspondence relates two formalisms in which the natural notions of reduction are confluent, this is not what defines the correspondence. We believe it is simply the notion of proofs as programs, formulas as types, proof normalisation as reductions, which describes the essence of the correspondence. There is nothing here to say that such reductions must be confluent; merely that the notions of reduction on proofs and terms should coincide[8]. Since, in this work, we take the logic as the starting point, it seems already evident from Gentzen's cut elimination that confluence is not a property we should insist upon. Although this point of view differs from much of the early work on Curry-Howard for classical logic (most notably in the natural deduction paradigm [43, 66, 68]), there is an increasing volume of literature which aims to study a 'full' notion of reduction for classical logic, and (at least initially) abandons confluence as a necessary criterion [10, 21, 92, 56, 98, 81].

The absence of confluence has implications for simulation results between calculi. In a confluent setting, if one wishes to compare reductions $M \to N$ in one calculus with reductions $P \to' Q$ in another, using some interpretation $[\![M]\!]$ from the first to the second, it is common to prove a statement such as "If $M \to N$ then there exists P with $[\![M]\!] \to' P$ and $[\![N]\!] \to' P$". Such a statement does not show simulation directly, but only simulation up to "joinability" in the target calculus (which is normally easier

[8] We have also heard it said that the eta laws of the λ-calculus must hold for a Curry-Howard Correspondence to be claimed. However, we do not regard this to be an essential feature of what the correspondence means.

to achieve). In a confluent calculus this can be justified by the notion that terms with a common reduct are 'essentially' the same, and will in particular share the same unique normal form. In a non-confluent setting, however, this statement is not very strong, because the extra reductions used to reach P may make essential choices (losing normal forms). For these reasons, we believe that it is only acceptable to claim that a non-confluent calculus can simulate another by proving a statement of the form: "If $M \to N$ then $[\![M]\!] \to' [\![N]\!]$".

Chapter 3

A Term Calculus for Classical Sequent Calculus

3.1 Overview

In this chapter we will define the $\mathcal{X}^i$-calculus, which is an untyped term calculus based on a Curry-Howard Correspondence with classical sequent calculus. It is named after the $\mathcal{X}$-calculus of [98], but is presented with propagation of cuts as an 'implicit' operation (as we will describe). It is essentially an *untyped* variant of one of the term representations for sequent calculus proofs used in Urban's PhD [92], and the reduction behaviour is that described by Urban's cut elimination. The notation we use is not based on Urban's prefix notation, but rather the infix notation of [98], since this makes more explicit the input/output symmetry of the calculus, in particular in the case of cuts.

Urban's cut elimination procedures were designed to be close to the original cut elimination of Gentzen, but to allow the propagation of cuts over other cuts (which is necessary for, e.g., simulation of beta reduction from the λ-calculus [92]) and to avoid as much as possible restricting the inherent non-determinism of the system, while retaining a strongly normalising set of reductions. Since, as we discussed in the introduction, we regard the non-confluence of cut-elimination to be a natural property of proof reductions for classical logic, this forms an excellent basis for our work. We choose to have cut propagation as a meta-operation (rather than one whose step-by-step definition forms part of the reduction rules of the calculus [98]), mainly because this aids comparisons with other calculi later on. It also emphasises the separation of propagation behaviour from the reduction of logical cuts, which we find philosophically useful.

3.2 The $\mathcal{X}^i$-Calculus

Since the sequents of classical sequent calculus have multiple formulas on *both* sides of the sequent, when defining a term inhabitation for the logic it is natural to have two alphabets of names to index them. Following the definition of the $\mathcal{X}$-calculus (Section 2.9), we see those names indexing formulas on the left of the sequent as inputs, and call them *sockets*, and those on the right as outputs, and call them *plugs*. This particular calculus is chosen to correspond with a logic with the two connectives negation ($\neg$) and implication ($\rightarrow$) as primitive (which form a *complete* set of connectives, in terms of logical expressibility).

Definition 3.2.1 ($\mathcal{X}^i$-Terms (cf. Definition 2.9.1))**.** *The terms of the $\mathcal{X}^i$-calculus (ranged over by P,Q,R, etc.) are defined by the following syntax, where x, y range over the infinite set of* sockets *and α, β over the infinite set of* plugs *(sockets and plugs together form the set of* connectors*).*

$$P, Q ::= \underset{\text{capsule}}{\langle x.\alpha \rangle} \mid \underset{\text{export}}{\widehat{x}P\widehat{\alpha}\cdot\beta} \mid \underset{\text{import}}{P\widehat{\alpha}\,[y]\,\widehat{x}Q} \mid \underset{\text{not-right}}{\widehat{x}P\cdot\alpha} \mid \underset{\text{not-left}}{x\cdot P\widehat{\alpha}} \mid \underset{\text{cut}}{P\widehat{\alpha}\,\dagger\,\widehat{x}Q}$$

In order to understand the reduction behaviour of this calculus, and also to aid various discussions later on, it is useful to be able to describe the location of occurrences of a free connector in a term. To this end, we make use of the following definitions:

Definition 3.2.2 (Exhibiting and Introducing a Connector)**.** *For any $\mathcal{X}^i$-term P and socket x, we say P* exhibits *x if there is an occurrence of x at the top-level of P's syntactic structure.*

We say that P introduces *x if $x \in fs(P)$ but, for all proper subterms P' of P, $x \notin fs(P')$ (alternatively, x occurs* uniquely *at the top-level of P's syntactic structure).*

For example, $\widehat{x}\langle x.\alpha\rangle\widehat{\beta}\cdot\alpha$ exhibits α but does not introduce α (since there is a further occurrence of α within a subterm). Also, $\langle x.\alpha\rangle$ introduces (and therefore exhibits) both x and α.

The most important use for these definitions is in understanding the behaviour of the cut-elimination procedure. A cut $P\widehat{\alpha}\,\dagger\,\widehat{x}Q$ in which P introduces α and Q introduces x can always be removed (and possibly replaced by new cuts between the subterms of P and Q) - this is the ultimate goal of the cut elimination procedure. These rules (which are called the *logical* reduction rules) are described as follows.

Definition 3.2.3 (Logical Rules)**.** *The logical rules are presented by:*

$$
\begin{array}{lrll}
(\textit{cap}): & \langle y.\alpha\rangle\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle & \to \langle y.\beta\rangle & \\
(\textit{impR}): & (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}\langle x.\gamma\rangle & \to \widehat{y}P\widehat{\beta}\cdot\gamma & \alpha \notin \textit{fp}(P) \\
(\textit{impL}): & \langle y.\alpha\rangle\widehat{\alpha} \dagger \widehat{x}(P\widehat{\beta}\,[x]\,\widehat{z}Q) & \to P\widehat{\beta}\,[y]\,\widehat{z}Q & x \notin \textit{fs}(P,Q) \\
(\textit{not-right}): & (\widehat{y}P\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle & \to \widehat{y}P\cdot\beta & \alpha \notin \textit{fp}(P) \\
(\textit{not-left}): & \langle y.\alpha\rangle\widehat{\alpha} \dagger \widehat{x}(x\cdot P\widehat{\beta}) & \to y\cdot P\widehat{\beta} & x \notin \textit{fs}(P) \\
(\textit{imp}): & (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R) & \to \left\{\begin{array}{l} Q\widehat{\gamma} \dagger \widehat{y}(P\widehat{\beta}\dagger\widehat{z}R) \\ (Q\widehat{\gamma}\dagger\widehat{y}P)\widehat{\beta}\dagger\widehat{z}R \end{array}\right\} & \begin{array}{l}\alpha \notin \textit{fp}(P),\\ x \notin \textit{fs}(Q,R)\end{array} \\
(\textit{not}): & (\widehat{y}P\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}(x\cdot Q\widehat{\beta}) & \to Q\widehat{\beta}\dagger\widehat{y}P & \alpha \notin \textit{fp}(P), x \notin \textit{fs}(Q)
\end{array}
$$

The logical rules are *only* applicable in the special case of a cut whose subterms both introduce the appropriate connector. In all other cases, a cut is reduced by 'seeking out' the positions in its subterms where the appropriate connectors are exhibited. For example, if P does not introduce α, then a cut $P\widehat{\alpha}\dagger\widehat{x}Q$ can be reduced by pushing copies of the cut with Q through the structure of P, depositing a cut at the level of each occurrence of α in P. A similar behaviour is possible when x is not introduced in Q. This reduction behaviour is referred to as (*left-* or *right-*)*propagation*.

In contrast to the $\mathcal{X}$-calculus, we present propagation as a meta-operation, external to the calculus itself, in much the same way as substitution is treated in the λ-calculus[1]. We introduce the notation $P\{\alpha \leftarrow\!\!\!\rightarrow \widehat{x}Q\}$ to denote the result of left-propagation, which propagates through the structure of the term P, connecting each occurrence of α with a new cut with Q, via x. The notation $Q\{P\widehat{\alpha} \leftarrow\!\!\!\rightarrow x\}$ is used for the analogous right-propagation operation. Note that this notation is not a part of the syntax of the calculus; rather it denotes the *result* of evaluating the associated operations. These are defined as follows:

Definition 3.2.4 (Propagation Operations)**.** *The operation $P\{\alpha \leftarrow\!\!\!\rightarrow \widehat{x}Q\}$ is defined recur-*

[1] $\mathcal{X}$ can be seen as the 'explicit' (i.e., propagation is included explicitly in the reduction rules) version of the $\mathcal{X}^i$ calculus, just as $\lambda\mathbf{x}$ can be seen as the 'explicit' version of the λ-calculus. In terms of Urban's work, the $\mathcal{X}^i$ calculus is essentially the untyped version of his $\to_{aux}$ cut elimination procedure, while $\mathcal{X}$ can be equally compared with the 'localised version', $\to_{loc}$ [92].

sively over the structure of P, as follows:

$$\begin{aligned}
\langle y.\alpha\rangle\{\alpha \leftrightarrow \widehat{x}P\} &= \langle y.\alpha\rangle\widehat{\alpha} \dagger \widehat{x}P \\
\langle y.\beta\rangle\{\alpha \leftrightarrow \widehat{x}P\} &= \langle y.\beta\rangle && \beta \neq \alpha \\
(\widehat{y}Q\widehat{\beta}\cdot\alpha)\{\alpha \leftrightarrow \widehat{x}P\} &= (\widehat{y}(Q\{\alpha \leftrightarrow \widehat{x}P\})\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}P \\
(\widehat{y}Q\widehat{\beta}\cdot\gamma)\{\alpha \leftrightarrow \widehat{x}P\} &= \widehat{y}(Q\{\alpha \leftrightarrow \widehat{x}P\})\widehat{\beta}\cdot\gamma, && \gamma \neq \alpha \\
(Q\widehat{\beta}\,[z]\,\widehat{y}R)\{\alpha \leftrightarrow \widehat{x}P\} &= (Q\{\alpha \leftrightarrow \widehat{x}P\})\widehat{\beta}\,[z]\,\widehat{y}(R\{\alpha \leftrightarrow \widehat{x}P\}) \\
(\widehat{y}Q\cdot\alpha)\{\alpha \leftrightarrow \widehat{x}P\} &= (\widehat{y}(Q\{\alpha \leftrightarrow \widehat{x}P\})\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}P \\
(\widehat{y}Q\cdot\gamma)\{\alpha \leftrightarrow \widehat{x}P\} &= \widehat{y}(Q\{\alpha \leftrightarrow \widehat{x}P\})\cdot\gamma, && \gamma \neq \alpha \\
(z\cdot Q\widehat{\beta})\{\alpha \leftrightarrow \widehat{x}P\} &= z\cdot(Q\{\alpha \leftrightarrow \widehat{x}P\})\widehat{\beta} \\
(Q\widehat{\beta} \dagger \widehat{y}\langle y.\alpha\rangle)\{\alpha \leftrightarrow \widehat{x}P\} &= (Q\{\alpha \leftrightarrow \widehat{x}P\})\widehat{\beta} \dagger \widehat{x}P \\
(Q\widehat{\beta} \dagger \widehat{y}R)\{\alpha \leftrightarrow \widehat{x}P\} &= (Q\{\alpha \leftrightarrow \widehat{x}P\})\widehat{\beta} \dagger \widehat{y}(R\{\alpha \leftrightarrow \widehat{x}P\}), && R \neq \langle y.\alpha\rangle
\end{aligned}$$

The operation $Q\{P\widehat{\alpha} \leftrightarrow x\}$ is defined recursively over the structure of Q, as follows:

$$\begin{aligned}
\langle x.\beta\rangle\{P\widehat{\alpha} \leftrightarrow x\} &= P\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle \\
\langle y.\beta\rangle\{P\widehat{\alpha} \leftrightarrow x\} &= \langle y.\beta\rangle, && y \neq x \\
(\widehat{y}Q\widehat{\beta}\cdot\gamma)\{P\widehat{\alpha} \leftrightarrow x\} &= \widehat{y}(Q\{P\widehat{\alpha} \leftrightarrow x\})\widehat{\beta}\cdot\gamma \\
(Q\widehat{\beta}\,[x]\,\widehat{y}R)\{P\widehat{\alpha} \leftrightarrow x\} &= P\widehat{\alpha} \dagger \widehat{x}((Q\{P\widehat{\alpha} \leftrightarrow x\})\widehat{\beta}\,[x]\,\widehat{y}(R\{P\widehat{\alpha} \leftrightarrow x\})) \\
(Q\widehat{\beta}\,[z]\,\widehat{y}R)\{P\widehat{\alpha} \leftrightarrow x\} &= (Q\{P\widehat{\alpha} \leftrightarrow x\})\widehat{\beta}\,[z]\,\widehat{y}(R\{P\widehat{\alpha} \leftrightarrow x\}), && z \neq x \\
(\widehat{y}Q\cdot\gamma)\{P\widehat{\alpha} \leftrightarrow x\} &= y\cdot(Q\{P\widehat{\alpha} \leftrightarrow x\})\widehat{\gamma} \\
(x\cdot Q\widehat{\beta})\{P\widehat{\alpha} \leftrightarrow x\} &= P\widehat{\alpha} \dagger \widehat{x}(x\cdot(Q\{P\widehat{\alpha} \leftrightarrow x\})\widehat{\beta}) \\
(z\cdot Q\widehat{\beta})\{P\widehat{\alpha} \leftrightarrow x\} &= z\cdot(Q\{P\widehat{\alpha} \leftrightarrow x\})\widehat{\beta}, && z \neq x \\
(\langle x.\beta\rangle\widehat{\beta} \dagger \widehat{y}R)\{P\widehat{\alpha} \leftrightarrow x\} &= P\widehat{\alpha} \dagger \widehat{y}(R\{P\widehat{\alpha} \leftrightarrow x\}) \\
(Q\widehat{\beta} \dagger \widehat{y}R)\{P\widehat{\alpha} \leftrightarrow x\} &= (Q\{P\widehat{\alpha} \leftrightarrow x\})\widehat{\beta} \dagger \widehat{y}(R\{P\widehat{\alpha} \leftrightarrow x\}), && Q \neq \langle x.\beta\rangle
\end{aligned}$$

The propagation operations are used to define the two *propagation rules* for this calculus.

Definition 3.2.5 (Propagation Rules)**.** *We define two* cut-propagation *rules.*

$$\begin{aligned}
(\textit{prop}\text{-L}) &: P\widehat{\alpha} \dagger \widehat{x}Q \to P\{\alpha \leftrightarrow \widehat{x}Q\} \textit{ if } P \textit{ does not introduce } \alpha \\
(\textit{prop}\text{-R}) &: P\widehat{\alpha} \dagger \widehat{x}Q \to Q\{P\widehat{\alpha} \leftrightarrow x\} \textit{ if } Q \textit{ does not introduce } x
\end{aligned}$$

Hereafter, we will write $\to$ for the reflexive, transitive, compatible reduction relation generated by the logical and propagation rules.

We can prove a number of results about propagation. The first two points below assert that connectors which are 'sought out' by propagation (i.e., the α in $Q\{\alpha \leftrightarrow \widehat{x}P\}$ and the x in $P\{Q\widehat{\alpha} \leftrightarrow x\}$), never occur in the resulting terms. The other parts show that the connectors introduced in a term are preserved under propagation, provided they are neither 'sought'

by the propagation operation, nor paired in a capsule with the connector which is 'sought'. These will be useful to us for the proofs in Chapter 7.

Proposition 3.2.6 (Effects of Propagation)**.** *Let P,Q be arbitrary $\mathcal{X}^i$-terms. Then, for any $y \notin fs(P)$ and $\beta \notin fs(P)$, we have:*

1. $\alpha \notin fs(Q\{\alpha \leftrightarrow\!\triangleright \widehat{x}P\})$.
2. $x \notin fp(P\{Q\widehat{\alpha} \leftrightarrow\!\triangleright x\})$.
3. *Q introduces y, if and only if, either $Q = \langle y.\alpha \rangle$ or $Q\{\alpha \leftrightarrow\!\triangleright \widehat{x}P\}$ introduces y.*
4. *Q introduces β and $\beta \neq \alpha$, if and only if, $Q\{\alpha \leftrightarrow\!\triangleright \widehat{x}P\}$ introduces β.*
5. *Q introduces y and $y \neq x$, if and only if, $Q\{P\widehat{\alpha} \leftrightarrow\!\triangleright x\}$ introduces y.*
6. *Q introduces β if and only if, either $Q = \langle x.\beta \rangle$ or $Q\{P\widehat{\alpha} \leftrightarrow\!\triangleright x\}$ introduces β.*

Proof. By exhaustive case analysis of the structure of Q, using Definitions 3.2.4 and 3.2.2. □

3.2.1 Propagation and Strong Normalisation

Despite the many cases of Definition 3.2.4, almost all can be understood to push the operation throughout the syntactic structure of the term, depositing cuts in each place where the appropriate connector is introduced. However, for both operations, the penultimate case listed does not appear to fit this pattern. Indeed, the 'obvious' choice for propagating over a cut appears to be the final case listed (i.e., propagating directly into the subterms). These rules are introduced in [92], in which the special cases are explained to be the solutions to a technical obstacle in applying the desired technique of symmetric reducibility candidates, as used by Barbanera and Berardi in [10]. It is natural to wonder whether treating these special cases in this way is necessary, or is purely a technical consequence of the particular proof involved. Indeed, in [98], which is roughly based on the 'localised version' of the same cut elimination procedure, these special cases are not included; all propagation over cuts is treated as in the last cases above. However, we observe that the special cases are in fact, essential for strong normalisation of typeable terms, which was one of the fundamental goals of Urban's work, and our own.

We identify here an invariant of the reduction behaviour of this calculus, which is intuitively related to strong normalisation. As we have explained above, the 'purpose' of the propagation rules is to deposit cuts in positions where the connectors which they bind

are immediately exhibited. These positions are sought out by propagation, and copies of cuts are deposited at each. In these resulting cuts, the sought-out connector is now introduced in the subterm. Furthermore, during further reduction while this cut remains (i.e., is not reduced itself), the connector will *remain introduced in the subterm*. Therefore, in a sense, the work done by propagation is never undone; once these occurrences are found, the copies of cuts remain 'anchored' to the occurrences until they are reduced. Other propagation may take place on these terms, but, since different connectors will be sought out, the resulting new cuts will 'anchor' in different places. Actually, this is not quite true: if it were the case that each syntactic construct in a term could only 'anchor' one cut, then our invariant would seem reasonable. A capsule $\langle x.\alpha \rangle$ introduces *two* connectors (in fact, it is the only syntax construct to do so), and so it is possible for two different cuts to try and 'anchor' with it. If not avoided, this leads to an unfortunate source of loops in reduction, in which two cuts can be seen to fight for 'anchor position', as is illustrated by the following example.

Definition 3.2.7 (Counter-example to Strong Normalisation with naïve Propagation)**.** *If propagation over cuts were always dealt with by the last cases in Definition 3.2.4 (i.e., in the cases $*$ below), then it would be possible to make the following cyclic reduction sequence, for any $\mathcal{X}^i$-terms P and R such that $\beta \notin \mathit{fp}(P)$ and $x \notin \mathit{fs}(R)$:*

$$
\begin{array}{rll}
(P\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle)\widehat{\beta} \dagger \widehat{y}R & \to & (P\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle)\{\beta \leftarrow\!\!\triangleright \widehat{y}R\} \qquad (\mathit{prop}\text{-L}) \\
& = & (P\{\beta \leftarrow\!\!\triangleright \widehat{y}R\})\widehat{\alpha} \dagger \widehat{x}(\langle x.\beta\rangle\{\beta \leftarrow\!\!\triangleright \widehat{y}R\}) \quad (*) \\
& = & P\widehat{\alpha} \dagger \widehat{x}(\langle x.\beta\rangle\widehat{\beta} \dagger \widehat{y}R) \qquad (\textit{since } \beta \notin P) \\
& \to & (\langle x.\beta\rangle\widehat{\beta} \dagger \widehat{y}R)\{P\widehat{\alpha} \leftarrow\!\!\triangleright x\} \qquad (\mathit{prop}\text{-R}) \\
& \to & (\langle x.\beta\rangle\{P\widehat{\alpha} \leftarrow\!\!\triangleright x\})\widehat{\beta} \dagger \widehat{y}(R\{P\widehat{\alpha} \leftarrow\!\!\triangleright x\}) \quad (*) \\
& \to & (P\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle)\widehat{\beta} \dagger \widehat{y}R \qquad (\textit{since } x \notin R) \\
& \ldots &
\end{array}
$$

The conditions $\beta \notin \mathit{fp}(P)$ and $x \notin \mathit{fs}(R)$ are not necessary, but simplify the presentation of the problematic example. The example is avoided by the special cases in Definition 3.2.4; i.e., by the design choices of Urban. The special cases can be understood as 'shortcuts'; starting from any term of the form $(P\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle)\widehat{\beta} \dagger \widehat{y}R$ it is still possible to reduce either of the two cuts before the other, but if the outermost is chosen to be evaluated first, then rather than depositing the new cut as in $(P\{\beta \leftarrow\!\!\triangleright \widehat{y}R\})\widehat{\alpha} \dagger \widehat{x}(\langle x.\beta\rangle\widehat{\beta} \dagger \widehat{y}R)$ it is implicitly and immediately right-propagated, and reduced to $(P\{\beta \leftarrow\!\!\triangleright \widehat{y}R\})\widehat{\alpha} \dagger \widehat{x}(R\langle x/y\rangle)$ which, by α-conversion is reflected by the special case of Definition 3.2.4.

3.3 Type Assignment for $\mathcal{X}^i$

Since $\mathcal{X}^i$ is the untyped analogue of a typed term assignment for sequent proofs, it comes with a natural notion of type-assignment. The type system that we present in this section corresponds with a simple sequent calculus for the restriction of classical logic to the two connectives implication ($\rightarrow$) and negation ($\neg$). The sequent calculus on which the type system is based is a variant of Kleene's G3, in which structural rules are treated implicitly. Arbitrary weakenings are allowed at the leaves of a derivation (in the (ax) rules), while contraction is treated implicitly per rule; if a statement is introduced to a context in which it is already present, it is simply merged. Gentzen's original formulation also included *exchange* rules, for reordering the statements on the left and right of a sequent; in our setting we treat these collections of statements as (unordered) sets.

Definition 3.3.1 (Types and Contexts)**.**

1. *The set of* simple types $\mathcal{T}$*, ranged over by* A, B*, is defined over a set of* atomic types $\mathcal{V} = \{\varphi_1, \varphi_2, \varphi_3, \ldots\}$ *by the grammar:*

$$A, B ::= \varphi \mid \neg A \mid A \rightarrow B$$

2. *A* left context 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* $\Gamma \cup \{x{:}A\}$*. When writing a context as* $\Gamma, x{:}A$*, we indicate that either* Γ *is not defined on* x *or contains the same statement* $x{:}A$*. We write* $\Gamma \backslash x$ *(read as "*Γ without x*") for the context from which the statement concerning* x*, if any, has been removed.*

   Right contexts Δ*, and the notations* $\alpha{:}A, \Delta$ *and* $\Delta \backslash \alpha$ *are defined in a similar way.*

3. *A pair* $\langle \Gamma; \Delta \rangle$ *is usually referred to simply as a* context*, and is a shorthand for the sequent (with labelled formulas)* $\Gamma \vdash \Delta$*. We will sometimes also refer to left/right contexts simply as contexts, when it is clear to do so.*

Armed with these definitions, we can define the simple type assignment system for the calculus.

Definition 3.3.2 (Typing for $\mathcal{X}^i$)**.**

1. Type judgements *are expressed via a ternary relation* $P :\cdot \Gamma \vdash \Delta$*, where* Γ *is a left context,* Δ *is a right context, and* P *is an* $\mathcal{X}^i$*-term. We say that* P *is the* witness *of this judgement.*

2. Type assignment *is defined by the following sequent calculus:*

$$\frac{}{\langle x.\alpha\rangle :\cdot \Gamma, x:A \vdash \alpha:A, \Delta}\,(ax) \qquad \frac{P :\cdot \Gamma \vdash \alpha:A, \Delta \quad Q :\cdot \Gamma, x:A \vdash \Delta}{P\widehat{\alpha} \dagger \widehat{x}Q :\cdot \Gamma \vdash \Delta}\,(cut)$$

$$\frac{P :\cdot \Gamma, x:A \vdash \alpha:B, \Delta}{\widehat{x}P\widehat{\alpha}\cdot\beta :\cdot \Gamma \vdash \beta:A{\to}B, \Delta}\,(\to\mathcal{R}) \qquad \frac{P :\cdot \Gamma \vdash \alpha:A, \Delta \quad Q :\cdot \Gamma, y:B \vdash \Delta}{P\widehat{\alpha}\,[x]\,\widehat{y}Q :\cdot \Gamma, x:A{\to}B \vdash \Delta}\,(\to\mathcal{L})$$

$$\frac{P :\cdot \Gamma, x:A \vdash \Delta}{\widehat{x}P\cdot\alpha :\cdot \Gamma \vdash \alpha:\neg A, \Delta}\,(\neg\mathcal{R}) \qquad \frac{P :\cdot \Gamma \vdash \alpha:A, \Delta}{x\cdot P\widehat{\alpha} :\cdot \Gamma, x:\neg A \vdash \Delta}\,(\neg\mathcal{L})$$

We write $P :\cdot \Gamma \vdash \Delta$ *if there exists a derivation using the above rules that has this judgement in the bottom line.*

It is easy to show that a judgement $P :\cdot \Gamma \vdash \Delta$ includes types for (at least) the free connectors in P. In terms of the Curry-Howard Correspondence, P represents the syntactic structure of a proof of the sequent $\Gamma \vdash \Delta$, so P is in fact a witness to this sequent being provable in the underlying logic. Note that there is no notion of a type for P itself; rather, the whole context $\langle\Gamma;\Delta\rangle$ describes a consistent way of assigning types to P's connectors.

It is important to emphasise that the typing rules include a notion of implicit contraction; if a new statement is introduced on the bottom line of a rule, but it was already present in the context, then it is simply merged. We do not consider duplicate statements, as we consider contexts to be unordered sets. This also implies that a typing rule cannot be applied if it would result in the addition of a statement $x:A$ to a context Γ, say, in which x was already assigned a different type.

Example 3.3.3. *If the judgement* $P :\cdot x:A \vdash \alpha:B, \beta:A$ *had been derived, and one wished to apply the* (*impR*) *rule to this statement, binding the connectors* x *and* α*, it would not be possible for the connector exhibited in the premise to be* β*, since this would mean* β *was assigned both type* A *and type* $A{\to}B$*. Put more succinctly, the* $\mathcal{X}^i$*-term* $\widehat{x}\langle x.\beta\rangle\widehat{\alpha}\cdot\beta$ *is not typeable in the type system presented above.*

Since the $\mathcal{X}^i$-calculus is the untyped analogue of Urban's term annotation for sequent calculus proofs, if we omit the types from his syntax (as he does for convenience in the bulk

of his work), we can show how the two different term representations correspond. In order to facilitate the comparison, we will write Urban's "co-names" with Greek characters (rather than Roman characters from the first half of the alphabet, as he does). Explicitly then, the correspondence is as follows:

Definition 3.3.4 (Correspondence between $\mathcal{X}^i$-terms and Urban's term annotation for classical sequent calculus [92])**.**

Urban	$\mathcal{X}^i$
$\mathit{Ax}(x, \alpha)$	$\langle x.\alpha \rangle$
$\mathit{Cut}(<\alpha>P, (x)Q)$	$P\widehat{\alpha} \dagger \widehat{x}Q$
$\mathit{Imp}_R((x)<\alpha>P, \beta)$	$\widehat{x}P\widehat{\alpha} \cdot \beta$
$\mathit{Imp}_L(<\alpha>P, (y)Q, x)$	$P\widehat{\alpha}\,[x]\,\widehat{y}Q$
$\mathit{Not}_R((x)P, \alpha)$	$\widehat{x}P \cdot \alpha$
$\mathit{Not}_L(<\alpha>P, x)$	$x \cdot P\widehat{\alpha}$

Furthermore, the implicit propagation operations can be related as follows:

$P\{\alpha := (x)Q\}$	$P\{\alpha \leftrightarrow \widehat{x}Q\}$
$Q\{x := (\alpha)P\}$	$Q\{P\widehat{\alpha} \leftrightarrow x\}$

Note that we choose not to represent propagation in a manner reminiscent of substitution, since it is not a substitution operation. Rather, it deposits cuts whose structure matches the connectors and subterm mentioned between the $\{\ \}$ brackets.

We have the following result for the simple type system:

Theorem 3.3.5 (Witness Reduction)**.** *If $P :\cdot \Gamma \vdash \Delta$, and $P \to Q$, then $Q :\cdot \Gamma \vdash \Delta$.*

Proof. $\mathcal{X}^i$-terms to which types have been assigned correspond to sequent proofs and can be equivalently represented in the term calculus of Urban; this result then follows from the soundness of the cut elimination procedure of Urban [92]. □

We also have a strong normalisation property. Again, this is immediate from the work of Urban.

Theorem 3.3.6 (Strong Normalisation of $\mathcal{X}^i$)**.** *If $P :\cdot \Gamma \vdash \Delta$ then P is strongly normalising.*

Proof. $\mathcal{X}^i$-terms to which types have been assigned correspond to sequent proofs; this result then follows from the strong normalisation result of Urban [92]. □

3.3.1 Principal Typings

For the simple type system presented above, we can define an algorithm to compute *principal typings* (in the language of [104]), i.e., to provide an analogous result to that for the simple type assignment system for the λ-calculus. The algorithm takes as input an $\mathcal{X}^i$-term and either computes the principal typing of the term, if it is typeable at all, or else fails, indicating that the term is not typeable.

By "most general", we mean that all other possible typing contexts can be obtained by the operations of *substitution* and *weakening* (adding redundant information to the context). The operation of substitution is the usual one for Curry types, but we give the definitions here for completeness.

Definition 3.3.7 (Substitutions)**.**

1. *A* substitution S *is a (possibly empty) set of pairs* (φ, A) *where each* φ *is a distinct atomic type, and each* A *a type. The pair is meant to denote the replacement of occurrences of* φ *with* A*. Hence, as notational sugar, we write such pairs* $(\varphi{\mapsto}A)$*, read as "*φ *maps to* A*".*

2. *For any substitution* S *and type* A*, the* action *of* S *on* A*, written as the juxtaposition* $(S\ A)$ *is defined recursively as follows:*

$$\begin{aligned}(S\ \varphi) &\triangleq \left\{\begin{array}{ll} A & \textit{if } (\varphi{\mapsto}A) \in S \\ \varphi & \textit{otherwise}\end{array}\right\} \\ (S\ \neg A) &\triangleq \neg(S\ A) \\ (S\ A_1{\to}A_2) &\triangleq (S\ A_1) \to (S\ A_2)\end{aligned}$$

3. *For the special case where the set of pairs is empty we use a special symbol* id *and call this the* identity substitution*.*

4. *For any two substitutions* S_1 *and* S_2*, we define the* composition $S_2{\circ}S_1$ *(which is itself a substitution, and may be read informally as "*S_2 after S_1*") by the usual function composition: for any type variable* φ*, we define* $(S_2{\circ}S_1\ \varphi) = (S_2\ (S_1\ \varphi))$*.*

5. *We say two substitutions* S_1 *and* S_2 *are equal if they are identical as functions, i.e. if for all type variables* φ*,* $(S_1\ \varphi) = (S_2\ \varphi)$*.*

6. *We extend this definition to allow substitutions to act on contexts in the obvious way (i.e. the substitution is performed on each of the types in the context).*

7. *For any substitution S we define*[2] *the* domain *and* range *of S to be the sets of atomic types* $\mathit{dom}(S) = \{\varphi \mid (S\ \varphi) \neq \varphi\}$ *and* $\mathit{range}(S) = \{\varphi \mid \exists \varphi' \in \mathit{dom}(S), \varphi \in \mathit{atoms}(S\ \varphi')\}$.

Principal typings will be defined using Robinson's unification algorithm [80]. Unification is also extended to contexts of sockets (and identically for plugs) in the following definition:

Definition 3.3.8 (Unification)**.** *1. The algorithm unify [80] takes two types as arguments and returns a substitution (or fails: we do not model failure cases explicitly, but assume that if none of the definitions below apply then the algorithm terminates immediately with failure). It is defined as follows:*

$$\begin{aligned}
\mathit{unify}\ \varphi\ \varphi &= \mathit{id} \\
\mathit{unify}\ \varphi\ A &= (\varphi \mapsto A)\ \mathit{if}\ \varphi \notin A \\
\mathit{unify}\ A\ \varphi &= \mathit{unify}\ \varphi\ A \\
\mathit{unify}\ \neg A\ \neg B &= \mathit{unify}\ A\ B \\
\mathit{unify}\ A_1 \to A_2\ B_1 \to B_2 &= S_2 \circ S_1 \\
&\quad \mathit{where} \\
&\quad\quad S_1 = \mathit{unify}\ A_1\ B_1 \\
&\quad\quad S_2 = \mathit{unify}\ (S_1\ A_2)\ (S_1\ B_2)
\end{aligned}$$

2. *Unification is extended to contexts as follows (where $\emptyset$ denotes an empty context):*

$$\mathit{unifyContexts}\ \emptyset\ \Gamma_2 = id$$

$$\mathit{unifyContexts}\ (x : A, \Gamma_1)\ \Gamma_2 = \left\{ \begin{array}{ll}
\mathit{unifyContexts}\ \Gamma_1\ \Gamma_2 & \mathit{if}\ x \notin \Gamma_2 \\
S_2 \circ S_1 & \mathit{if}\ x : B \in \Gamma_2 \\
\quad \mathit{where} & \\
S_1 = \mathit{unify}\ A\ B & \\
S_2 = \mathit{unifyContexts}\ (S_1\ (\Gamma_1 \backslash x))\ (S_1\ (\Gamma_2 \backslash x)) &
\end{array} \right\}$$

Recall that for a well-formed context $(x : A, \Gamma)$, it does not automatically follow that x is not mentioned in Γ (so long as it is with the type A), which is the reason for explicitly removing x in the recursive call above.

We assume the classical soundness and completeness results for unification, along with their extension to contexts:

[2] with a slight abuse of standard terminology

Lemma 3.3.9 (Soundness and Completeness of Unification [80])**.**

1. *If unify $A\ B$ succeeds, yielding a substitution S_u, then $(S_u\ A) = (S_u\ B)$.*
2. *If there exists a substitution S such that $(S\ A) = (S\ B)$ then unify $A\ B$ succeeds, yielding a substitution S_u, and there exists a substitution S' such that $S = S' \circ S_u$.*
3. *If unifyContexts $\Gamma_1\ \Gamma_2$ succeeds, yielding a substitution S_u, then $(S_u\ \Gamma_1) = (S_u\ \Gamma_2)$.*
4. *If there exists a substitution S such that $(S\ \Gamma_1) = (S\ \Gamma_2)$ then unifyContexts $\Gamma_1\ \Gamma_2$ succeeds, yielding a substitution S_u, and there exists a substitution S' such that $S = S' \circ S_u$.*

Unification of contexts is required in order to compute principal typings for a term with more than one immediate subterm (i.e., for import and cut terms). In these cases, the two recursive calls to the algorithm will generate distinct contexts, but in the cases where the same connector is mentioned in both contexts, they must be made to agree on the types before the contexts can be 'merged'. We also require the following standard result:

Lemma 3.3.10 (Soundness of Substitution)**.**
If $P :\cdot \Gamma \vdash \Delta$ then for any substitution S, $P :\cdot (S\ \Gamma) \vdash (S\ \Delta)$.

Proof. By straightforward induction on the structure of the term P. □

Definition 3.3.11 (Principal contexts)**.** *The procedure pC $:: \mathcal{X}^i \to \langle \Gamma; \Delta \rangle$ is defined in Figure 3.1, where typeof $\mathtt{c}\ \Psi$ (with $\mathtt{c}$ a connector, and Ψ a context) returns A, if $\mathtt{c} : A \in \Psi$, or else* fresh*, i.e., an atomic type not used elsewhere.*

The following results state that this algorithm does indeed compute principal typings.

Theorem 3.3.12 (Soundness and Completeness of *pC*)**.**

1. *Soundness: If pC $(P) = \langle \Gamma, \Delta \rangle$, then $P :\cdot \Gamma \vdash \Delta$.*
2. *Completeness: If $P :\cdot \Gamma \vdash \Delta$, then there exist Γ_P and Δ_P, and a substitution S such that pC $(P) = \langle \Gamma_P, \Delta_P \rangle$, and $(S\ \Gamma_P) \subseteq \Gamma$ and $(S\ \Delta_P) \subseteq \Delta$.*

Proof. See Proof A.1.1 in Appendix A. □

3.4 Other Logical Connectives

We have chosen to treat the two connectives implication and negation as the (only) primitive connectives in the logic underlying the $\mathcal{X}^i$-calculus. It is natural to ask whether the choice of extra, different, or fewer connectives would make a significant difference to the resulting calculus. One obvious observation to make in the setting of classical logic (in contrast with intuitionistic logic) is that many of the logical connectives are definable in terms of others. In fact, there are many possible choices of small subsets of the binary connectives which are *complete*, in the sense that any formula expressible using any other logical connectives is equivalent to a formula written using only connectives from the subset. It is easy to check that negation and implication form a complete set of connectives, in this sense. It is not the smallest such set, since (for example) the 'nand' connective forms a complete set on its own. However, we were also guided by the fact that certain connectives have a more-intuitive computational meaning than others. In particular, implication is an obvious choice, since the original Curry-Howard Correspondence identifies its computational content with functions, and it has an inherent role in the type systems for λ-calculi and functional programming languages. However, implication on its own is not complete for classical logic, and leads to a calculus in which the full symmetries of the logic cannot be exploited. For example, using implication alone, it is impossible to prove a sequent with an empty right-hand side, although many sequents with empty left-hand sides are provable. Once negation is added, the full symmetry of the system is restored (in particular, sequents with empty right-hand sides are derivable, which correspond to a proof that the assumptions on the right of the sequent are contradictory).

In [75] (coauthored with Jayshan Raghunandan), the implications of different choices of primitive connectives for a programming calculus based on cut-elimination, were explored. It was found that the idea of logical expressiveness does not coincide under the Curry-Howard Correspondence with computational expressiveness, in the sense that although there may be formulas (types) in one calculus which do not have logically equivalent formulas (types) in another, it may still be possible for the *terms* of the first calculus to be encodeable in the syntax of the second, in such a way that reductions and typeability (but not the exact typings) are preserved. In the language of that paper, 'computational expressiveness' does not imply 'logical expressiveness'. Conversely, even when one works with a complete set of connectives, one needs to be careful when encoding a calculus based on other connectives in order to preserve reductions. Often, the natural encodings from the logical point of view do *not* preserve reductions. We do not elaborate on this point here, since we are satisfied with the expressiveness of the two connectives we have chosen. Their computational meaning will be more-fully explored in Chapter 5.

In the case of more esoteric connectives, it is not always clear how to define appropriate logical cut elimination rules. For example, defining suitable cut elimination rules for a calculus based on the 'if and only if' ($\leftrightarrow$) connective is non-trivial [75]. The forthcoming PhD thesis of Raghunandan [74] gives a detailed approach to the definition of term calculi based on arbitrary connectives in the setting of classical sequent calculus.

Note that in the presence of multiple logical connectives, there is a possibility of "stuck" configurations in the calculus; terms can contain cuts but yet be in a form where no reduction rules are applicable to them. For example, the term $(\widehat{x}\langle x.\alpha\rangle\widehat{\alpha}\cdot\beta)\widehat{\beta} \dagger \widehat{y}(y \cdot \langle z.\gamma\rangle\widehat{\gamma})$ cannot be reduced. There is no cut-elimination rule corresponding to this case, since the introduction of β is by a rule corresponding to implication in the logic, and the introduction of y corresponds to a rule for negation. Such a cut does not exist in the logic (the formula in the cut would need to have both implication and negation as its principal connective, which is impossible), and so the term is guaranteed to be untypeable. However, this 'clash' of connectives is a different source of untypeability from the usual failures of unification in type assignment (which are due to the "occurs check": cf. the condition "*if* $\varphi \notin A$" in Definition 3.3.8). Note that these 'stuck' configurations can never occur in typeable terms, and so this does not affect our strong normalisation result. In particular, a typeable term is guaranteed to always run to a cut-free term, which is itself typeable (Theorems 3.3.5 and 3.3.6).

3.5 Confluent Restrictions

The $\mathcal{X}^i$-calculus as presented above is highly non-confluent. This is very much intentional; since our reduction rules implement Christian Urban's cut elimination (in the typeable case). In his thesis [92], this is a stated aim: "...the cut-elimination procedure should *not* restrict the collection of normal forms reachable from a given proof in such a way that 'essential' normal forms are no longer reachable." On the other hand, for certain practical applications (for example, if a programming language were to be built on top of the calculus), it is desirable to have a confluent reduction system. In particular, it is not possible to define a deterministic evaluation order on a non-confluent calculus without fundamentally changing (and restricting) its reduction behaviour. Although we are strongly in favour of studying the more-natural non-confluent reduction system in general, it is possible to restrict the system to obtain a confluent one when desired.

Two simple ways in which this can be achieved, are essentially defined by giving priority to the left and right propagation of cuts, when both alternatives are possible.

Definition 3.5.1 (Restrictions of $\mathcal{X}^i$-reduction)**.** *The call-by-name (CBN) and call-by-value restrictions of $\mathcal{X}^i$ are defined as follows:*

CBN: *Cuts may only be left-propagated if they cannot be right-propagated. I.e., the rule* $(prop\text{-L})$ *is replaced by the variant:*

$$(prop\text{-L}_{cbn}) : P\widehat{\alpha} \dagger \widehat{x}Q \rightarrow P\{\alpha \leftarrow\!\!\triangleright \widehat{x}Q\} \text{ if } P \text{ does not introduce } \alpha, Q \text{ introduces } x$$

CBV: *Cuts may only be right-propagated if they cannot be left-propagated. I.e., the rule* $(prop\text{-R})$ *is replaced by the variant:*

$$(prop\text{-R}_{cbv}) : P\widehat{\alpha} \dagger \widehat{x}Q \rightarrow Q\{P\widehat{\alpha} \triangleleft\!\!\rightarrow x\} \text{ if } Q \text{ does not introduce } x, P \text{ introduces } \alpha$$

These are referred to as the call-by-name and call-by-value reduction strategies of the calculus [56, 98]. Lengrand [56] proves the confluence of variants of these strategies. The main differences between Lengrand's system and ours (apart from notational issues) are that we inhabit a logic extended with negation as well as implication, and in his system the analogous rule to our (imp) rule is restricted to only one of the two possible reducts, per strategy. This restriction is necessary for his proof of confluence, but it is not known whether if one leaves the rule unrestricted (but imposes one of the restrictions on cut propagation above), this results in a confluent set of reductions. For the purposes of this work (in which confluence is not a major feature) we will content ourselves with the restrictions in the above definitions when we wish to discuss call-by-name or call-by-value notions of reduction.

3.6 Summary

The $\mathcal{X}^i$-calculus introduced in this chapter will be used as the basis of those aspects of this thesis which are concerned with the paradigm of classical sequent calculus. In the next chapter, it will be used to study the problem of introducing ML-style *shallow polymorphism* in the setting of a term calculus based on classical logic. Later on, in Chapter 7, we will compare the $\mathcal{X}^i$-calculus with a term calculus based on classical natural deduction, and attempt to relate the two paradigms.

$$
\begin{aligned}
\mathit{pC}\,(\langle x.\alpha\rangle) &= \langle x{:}\varphi; \alpha{:}\varphi\rangle \\
\quad \textit{where } \varphi &= \textit{fresh} \\
\mathit{pC}\,(\widehat{x}P\widehat{\alpha}\cdot\beta) &= \langle \Gamma\backslash x; (\Delta\backslash\alpha)\cup\beta : A\to B\rangle, && \textit{if } \beta\notin\Delta \\
&= S\,\langle \Gamma\backslash x; \Delta\backslash\alpha\rangle, && \textit{if } \beta{:}C\in\Delta \\
\quad \textit{where } \langle\Gamma;\Delta\rangle &= \mathit{pC}\,(P) \\
A &= \textit{typeof}\; x\;\Gamma \\
B &= \textit{typeof}\; \alpha\;\Delta \\
S &= \textit{unify}\; C\; A\to B \\
\mathit{pC}\,(P\widehat{\alpha}\,[y]\,\widehat{x}Q) &= S_2\circ S_1\,\langle \Gamma_P\cup(\Gamma_Q\backslash x)\cup y{:}A\to B; (\Delta_P\backslash\alpha)\cup\Delta_Q\rangle \\
&\qquad\qquad \textit{if } y\notin (S_2\circ S_1\;\Gamma_P\cup(\Gamma_Q\backslash x)) \\
&= S_3\circ S_2\circ S_1\,\langle \Gamma_P\cup(\Gamma_Q\backslash x); (\Delta_P\backslash\alpha)\cup\Delta_Q\rangle, \\
&\qquad\qquad \textit{if } y{:}C\in (S_2\circ S_1\;\Gamma_P\cup(\Gamma_Q\backslash x)) \\
\quad \textit{where } \langle\Gamma_P;\Delta_P\rangle &= \mathit{pC}\,(P) \\
\langle\Gamma_Q;\Delta_Q\rangle &= \mathit{pC}\,(Q) \\
A &= \textit{typeof}\; \alpha\;\Delta_P \\
B &= \textit{typeof}\; x\;\Gamma_Q \\
S_1 &= \textit{unifyContexts}\; \Gamma_P\;(\Gamma_Q\backslash x) \\
S_2 &= \textit{unifyContexts}\; (S_1\;\Delta_P\backslash\alpha)\;(S_1\;\Delta_Q) \\
S_3 &= \textit{unify}\; C\;(S_2\circ S_1\; A\to B) \\
\mathit{pC}\,(x\cdot P\widehat{\alpha}) &= S\,\langle \Gamma_P, x:B; \Delta_P\backslash\alpha\rangle, \\
\quad \textit{where } \langle\Gamma_P;\Delta_P\rangle &= \mathit{pC}\,(P) \\
A &= \textit{typeof}\; \alpha\;\Delta_P \\
B &= \textit{typeof}\; x\;\Gamma_P \\
S &= \textit{unify}\; \neg A\; B \\
\mathit{pC}\,(\widehat{x}P\cdot\alpha) &= S\,\langle \Gamma_P\backslash x; \Delta_P, \alpha:B\rangle, \\
\quad \textit{where } \langle\Gamma_P;\Delta_P\rangle &= \mathit{pC}\,(P) \\
A &= \textit{typeof}\; x\;\Gamma_P \\
B &= \textit{typeof}\; \alpha\;\Delta_P \\
S &= \textit{unify}\; \neg A\; B \\
\mathit{pC}\,(P\widehat{\alpha}\,\dagger\,\widehat{x}Q) &= S_3\circ S_2\circ S_1\,\langle \Gamma_P\cup\Gamma_Q\backslash x; (\Delta_P\backslash\alpha)\cup\Delta_Q\rangle \\
\quad \textit{where } \langle\Gamma_P;\Delta_P\rangle &= \mathit{pC}\,(P) \\
\langle\Gamma_Q;\Delta_Q\rangle &= \mathit{pC}\,(Q) \\
A &= \textit{typeof}\; \alpha\;\Delta_P \\
B &= \textit{typeof}\; x\;\Gamma_Q \\
S_1 &= \textit{unifyContexts}\; \Gamma_P\;\Gamma_Q\backslash x \\
S_2 &= \textit{unifyContexts}\; (S_1\;\Delta_P\backslash\alpha)\;(S_1\;\Delta_Q) \\
S_3 &= \textit{unify}\; (S_2\circ S_1\; A)\;(S_2\circ S_1\; B)
\end{aligned}
$$

Figure 3.1: The principal contexts algorithm

Chapter 4

Polymorphism

4.1 Overview

Polymorphism is a powerful aspect of most modern programming languages. It is a mechanism for allowing a program to be applied in various contexts which each expect different types, and allows flexibility and reuse of code. In a non-polymorphic programming language for example, even if a function's behaviour is independent of the type of its argument, it must be redefined for each such type.

For a simple example, take the identity function (which takes one argument and returns it unchanged). This would be expressed in the λ-calculus as $\lambda x.x$, and in the $\mathcal{X}^i$-calculus by the term $\widehat{x}\langle x.\alpha\rangle\widehat{\alpha}\cdot\beta$. In either calculus, it is possible to derive the type $A{\rightarrow}A$ for this function (in the case of $\mathcal{X}^i$, the type is for the single output plug β, there being no notion of type for the term as a whole), for any Curry type A we choose. However, this type A must be fixed in a non-polymorphic setting, and so it would be impossible to type a program in which the function were applied to both an integer and a list, as A would need to be two different types.

When polymorphism is introduced, the identity function typically can be typed with the type $\forall X.(X{\rightarrow}X)$, where the $\forall$-bound type variable X ranges over all types. This then correctly expresses that the identity function may be typed as $A{\rightarrow}A$ for any and all formulas A. The rules for type-assignment typically allow this type to be *instantiated* several different times, so that (for example) it would be acceptable for the identity function to be applied to both an integer and a list in the same program. The type assignment rules employed to permit this style of polymorphism are based on logical inference rules concerning (second-order) quantifiers (in this case, $\forall$).

In this chapter we investigate how to adapt various notions of polymorphism to the $\mathcal{X}^i$-calculus. The problems here are two-fold: firstly what difference does the extension to a classical logic setting make, and secondly, how should polymorphism be implemented in the unusual setting of the sequent calculus? We choose to concentrate on the sequent calculus setting here, since not only does it allow us to tackle both questions, but it also provides a clearer understanding of issues involved. We show how the main results of this chapter can be adapted back to the natural deduction paradigm in Chapter 7.

The notions of polymorphism we discuss are those based on (second-order) logical quantifiers (e.g. $\forall$). Various other notions of polymorphism exist (for example, finitary polymorphism based on intersection types), but we focus on systems with quantifiers since they are naturally formalised using standard logical inference rules, which fits with the logic-based approach of this thesis.

We concentrate in this chapter on 'shallow polymorphism', focusing on the Hindley-Milner type system (famously used as the basis of the type system for ML), and discuss how this can be adapted to the $\mathcal{X}^i$-calculus. We show that a naïve approach to this problem results in an unsound type system, and show how the type system can be adapted in a novel way to avoid this unsoundness[1]. We also show that we can define a notion of principal types similar to the well-known result for ML.

We show that a 'dual' notion of shallow polymorphism can be employed, using existential quantification ($\exists$) rather than universal. Although in a programming setting based on intuitionistic logic (as λ-calculus and indeed ML can be seen to be) the addition of existential quantification does not make any new programs typeable, in a classical logic setting there are programs which can be made typeable in this way. We discuss how to understand the role of these two kinds of quantification with respect to the reduction rules. We show that the previous results adapt easily to this alternative system.

Finally, we discuss the idea of a type system combining *both* kinds of quantification at once. This system can type even more programs, and the roles of the quantifiers can be seen to be complimentary. We give an example of a program which is made typeable by employing this richer system. We discuss why it appears to be very difficult to find a principal typing algorithm for this richer type system, and present an idea for an approach to this problem.

This chapter builds on and corrects the work presented in [89]. The work presented in this chapter up to and including subsection 4.3.1 is based largely on this publication. The

[1] Dougherty et. al. have considered the similar problem of defining suitable notions of intersection (and union) type assignment for a calculus based on classical logic [29, 30]. It is interesting to note that these attempts have also been recently shown to be unsound, suggesting that polymorphism based on intersection types is also surprisingly subtle in the presence of classical logic [94, 95, 96].

remainder of the chapter is all unpublished work by the author.

4.1.1 Notation

Throughout this chapter, we will require various notation, particularly regarding the introduction of binders within our type language, and the handling of substitutions, renamings etc. within types.

The binders we will be employing within types come in the form of (second order) logical quantifiers: specifically $\forall$ (universal quantification) and, later, $\exists$ (existential quantification). We will usually refer to types containing these symbols as *quantified types*. We choose in this thesis to distinguish between bound and free 'type variables', using different notation and language to describe each. The 'free type variables' are (as in previous chapters) referred to as *atomic types*, and are represented by $\varphi, \varphi', \varphi_1, \varphi_2$ etc.. The *bound type variables* are chosen from the latter part of the uppercase Roman alphabet; i.e. W, X, Y, Z, X_1, X_2 etc.. We believe that maintaining a clear distinction between these two notions is both natural (since their meanings and behaviours are quite different) and useful. This is particularly so because the results of this chapter are technical in nature, and often depend on a careful treatment of quantified types.

The early part of the uppercase Roman alphabet ($A, B, C, D, E, F, A', A_1$ etc.) is still used to represent types. In order to describe quantified types separately from Curry types, we use the overlined version of this notation, i.e., the symbols $\overline{A}, \overline{B}, \overline{C}$ etc. Types with quantifiers are referred to (interchangeably) in this chapter as *generic types* or *type schemes*.

We often require operations to replace free atomic types φ with bound type variables X, and we write this operation as $\overline{A}[X/\varphi]$. We also require the replacement of bound type variables X with (Curry) types B, which we write as $\overline{A}[B/X]$. Note that these operations are kept distinct from Curry substitutions. We assume that these operations bind tighter than any logical connectives: for example, $\forall X.A[X/\varphi]$ should be read as $\forall X.(A[X/\varphi])$.

Since we will frequently be concerned with the question of which atomic types occur in which types, it is convenient to define the set *atoms*$(\overline{A})$ to be the set of all atomic types in a type $\overline{A}$ (note: this does not include bound type variables). We can then write $\varphi \in$ *atoms*$(\overline{A})$ to state that an atomic type occurs within a (generic) type $\overline{A}$. For convenience, we allow ourselves to write this as $\varphi \in \overline{A}$ when this does not cause confusion. We extend this notation to contexts in the obvious way, e.g., $\varphi \in \langle \Gamma; \Delta \rangle$ means that there exists a statement in $\langle \Gamma; \Delta \rangle$ featuring a type $\overline{A}$ such that $\varphi \in \overline{A}$.

When discussing generic types, i.e. types of the form $\forall X_1.\forall X_2.\ldots.\forall X_n.A$, we find it convenient to introduce the $\vec{\cdot}$ notation, e.g., $\overrightarrow{\forall X_i}.A$. We do not explicitly quantify over the subscripts, but it is always intended that a subscript i,j,k etc. is bound under the corresponding $\vec{\cdot}$. We extend this notation slightly informally to facilitate the statement and proof of our results, by using it as a shorthand for repetition in other statements. For example, we write $\{\overrightarrow{\varphi_i}\}$ for the set $\{\varphi_1, \varphi_2, \ldots\}$, and we write $\overrightarrow{\varphi_i \in \overline{A}}$ to mean "$\varphi_1 \in \overline{A}$ and $\varphi_2 \in \overline{A}$ etc.."

As usual, we assume all binders in types are α-converted appropriately to avoid clashes, and that all substitutions which cross binders are capture-avoiding.

4.2 Universal Shallow Polymorphism

In this chapter, we will be concerned with type systems based on the notion of *shallow polymorphism*, which employs quantifiers only on the outside of a type. In this section, we examine such type systems based on the addition of universal quantifiers (written $\forall$ and read 'for all') over (Curry) types. In a logical sense, these correspond to second-order quantifiers ranging over propositional formulas. The archetypal example of this paradigm is the Hindley-Milner [49, 58] type system, which underlies the type system for the ML programming language. The main advantages of this approach over that of System F [41, 78], for example, are practical: type checking and type assignment (within certain constraints, as we will explain) are decidable, and can be implemented by relatively straightforward algorithms [24]. In contrast, it has been shown that the corresponding problems are undecidable for System F [104].

We will first review the key aspects of the Hindley-Milner approach, and then examine how they can be brought to the more-general setting of the $\mathcal{X}^i$-calculus. This turns out to be non-trivial; not only do some aspects require extra machinery to be adapted naturally to the sequent calculus setting, but the intuitive approach fails for the general system; witness reduction is violated by the more-general reductions possible in the $\mathcal{X}^i$-calculus. This problem was not identified in the published work by the author [89], in which a witness reduction result for this type system is erroneously claimed. We examine here this problem, and identify three sufficient conditions for such a polymorphic type system to *fail* to be sound in this way. We compare with examples in the literature; in particular the unsoundness of ML when extended with various non-functional concepts, such as exceptions and control operators. By generalising a solution to these problems in the literature, we can give a restriction of our unsound type system, which is once more sound, and has a principal typing property in the spirit of that of ML.

We begin by examining the formal calculus on which ML is based.

4.2.1 The ML calculus

ML [58] is a language based upon an extension of the λ-calculus, which uses a different approach to System F for admitting polymorphism. To obtain decidability of type assignment, it permits only *shallow polymorphism*, which means that types are allowed to contain $\forall$ quantifiers only on the outside of their structure. For example, the formula $\forall X.(X \rightarrow X)$ would be a valid type in ML (which could be given to the identity function), whereas a type such as $(\forall X.X) \rightarrow (\forall Y.Y)$ would not (the $\forall$ symbols appear below the $\rightarrow$).

The usual syntax of the λ-calculus is extended with a new construct *let* $x = M$ $\texttt{in}$ N, which abstractly represents a substitution which has not yet taken place, in which M is to replace x inside the term N. This construct (along with its typing rule) is designed to give a workaround for the situation when an application $(\lambda x.N)\ M$ would be untypeable, whereas the reduct $N[M/x]$ can be typed - in this case the term *let* $x = M$ $\texttt{in}$ N may be used instead. The typing rule for *let* allows M to be given a shallow polymorphic type, and for this type to be used for x when trying to derive a type for N. With the addition of rules for *instantiating* types, it may be that several instances of the polymorphic type are used for different occurrences of x within N. This additional flexibility is what makes the system useful.

Definition 4.2.1 (ML Syntax)**.** *The syntax of* ML terms *is defined by:*

$$M, N ::= x \mid M\ N \mid \lambda x.M \mid \texttt{Fix}\, g.M \mid \textit{let}\ x = M\ \texttt{in}\ N$$

The construct $\texttt{Fix}\, g.M$ *is included to allow typeable recursion in the calculus. For simplicity in our discussions of polymorphism we choose to study the subset of ML expressions without* `Fix`*, and will hereafter only consider ML expressions within this subset.*

ML values *are defined by:* $V ::= x \mid \lambda x.M$

Definition 4.2.2 (ML Reductions)**.** *The reduction relation in ML is the transitive, compatible closure of the following two rules:*

$$\begin{aligned} (\lambda x.M)V &\rightarrow_{\text{ML}} M[V/x] \\ (\textit{let}\ x = V\ \texttt{in}\ M) &\rightarrow_{\text{ML}} M[V/x] \end{aligned}$$

As is clear from these reduction rules, the two terms *let* $x = V$ $\texttt{in}$ M and $(\lambda x.M)V$ both reduce to the same term $M[V/x]$ (semantically, these terms are interpreted in the

same way in [58]). However, they are treated differently by the type system. Milner writes, "... our aim is a type discipline which admits certain expressions in the first form and yet rejects their translations into the second form; this is because λ-abstractions may in general occur without an explicit operand, and need more careful treatment.". The typing rules for *let* provide the polymorphism in this system - it is allowed for each of the occurrences of x in M to be given a different instance of a polymorphic type found for V. This is in contrast to the usual way in which the term $(\lambda x.M)V$ would be treated, which would allow only *one* Curry type to be used for the variable x. These ideas are made formal in the following subsection.

4.2.2 Shallow Polymorphic Type Assignment for ML

We choose to present types and type assignment rules using the approach of Damas and Milner [24], as this gives a clearer treatment than that of [58].

Definition 4.2.3 (Type Schemes / Generic Types [24])**.** *The set of* type schemes *(also referred to as* generic types *in this work) is built from the usual Curry types by allowing any number (possibly zero) of $\forall$ quantifiers to be built on the outside. We will use A, B to range over the usual Curry types (extended with occurrences of bound variables), and $\overline{A}$ to range over type schemes, as defined below.*

$$\begin{aligned} A, B &::= \varphi \mid X \mid (A \to B) \\ \overline{A} &::= \forall X_1.\forall X_2.\ldots\forall X_n.A \ \ (n \geq 0) \end{aligned}$$

Note that in the case $n = 0$ in the definition of type schemes, we assert that any Curry type A is a type scheme itself. As in the discussions in the previous section, we distinguish between atomic types φ and type variable symbols X (whereas Milner chooses not to), and again consider only types with no free type variable symbols (e.g. occurrences of X which are not surrounded by $\forall X.$) to be well-formed. In fact, in [58] the set of Curry types is also extended with type constants, to represent concrete types such an integers, booleans, etc. However, this is more a practical consideration, and we leave them out in these discussions for simplicity[2].

We use the symbol Γ to represent a context of assumptions, as before. We write $\Gamma \vdash_{\text{ML}} M : \overline{A}$ to mean 'there is a type derivation assigning the (generic) type $\overline{A}$ to the term M under the context of assumptions Γ'. The form of these type derivations is defined as follows:

[2]Since, as we shall see, our type derivations are closed under substitution on atomic types, we can imagine extending these substitutions to replace atomic types with concrete types; everything works out in the same way.

Definition 4.2.4 ([24])**.** ML-type assignment *is defined by the following derivation rules.*

$$\frac{}{\Gamma, x:\overline{A} \vdash_{\text{ML}} x:\overline{A}}\,(ax) \qquad \frac{\Gamma \vdash_{\text{ML}} M:\overline{A} \quad \Gamma, x:\overline{A} \vdash_{\text{ML}} N:B}{\Gamma \vdash_{\text{ML}} \mathit{let}\ x = M\ \texttt{in}\ N:B}\,(\mathit{let})$$

$$\frac{\Gamma, x:A \vdash_{\text{ML}} M:B}{\Gamma \vdash_{\text{ML}} \lambda x.M:A{\to}B}\,(\to\mathcal{I}) \qquad \frac{\Gamma \vdash_{\text{ML}} M:A{\to}B \quad \Gamma \vdash_{\text{ML}} N:A}{\Gamma \vdash_{\text{ML}} M\ N:B}\,(\to\mathcal{E})$$

$$\frac{\Gamma \vdash_{\text{ML}} M:\overline{A}}{\Gamma \vdash_{\text{ML}} M:\forall X.\overline{A}[X/\varphi]}\,(\forall\mathcal{I})^{*} \qquad \frac{\Gamma \vdash_{\text{ML}} M:\forall X.\overline{A}}{\Gamma \vdash_{\text{ML}} M:\overline{A}\langle B/X\rangle}\,(\forall\mathcal{E})$$

* *if* φ *is not free in* Γ.

Notice that generic types $\overline{A}$ may not be used in the $(\to I)$ or $(\to E)$ rules - this reflects the fact that $\forall$-symbols may not appear inside an arrow type. In terms of the type assignment, this means that whenever an abstraction is to be typed, the variable abstracted over must have a fixed Curry type, just as in the original λ-calculus system. However, when x is a variable not occurring under an abstraction, the rules allow more freedom - if x has a polymorphic type in the context then the use of the (ax) and $(\forall E)$ rules allows a different instance of this type to be chosen each time x is used.

As an example, consider the term $(\lambda z.zz)(\lambda y.y)$. This remains untypeable in ML, just as it is in the λ-calculus, because z is bound in a lambda-abstraction over the self application zz. The self application requires z to be given two types, of the form $A{\to}B$ and A (for some Curry types A and B), whereas the lambda abstraction forces z to take a unique Curry type. Using the *let* construct, it is possible to form the term $\mathit{let}\ z = \lambda y.y\ \texttt{in}\ zz$, which will reduce in the same way as our original term. However, it is typeable in the ML system, because the type $\forall X.(X{\to}X)$ is derivable for $\lambda y.y$, and different instances of this type can be taken for the two occurrences of z (e.g. $(\varphi{\to}\varphi){\to}(\varphi{\to}\varphi)$ and $(\varphi{\to}\varphi)$ respectively):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y:\varphi' \vdash_{\text{ML}} y:\varphi'}\,(ax)}{\vdash_{\text{ML}} \lambda y.y:\varphi'{\to}\varphi'}\,(\to\mathcal{I})}{\vdash_{\text{ML}} \lambda y.y:\forall X.(X{\to}X)}\,(\forall\mathcal{I}) \quad \dfrac{\dfrac{}{z:\forall X.(X{\to}X) \vdash_{\text{ML}} z:\forall X.(X{\to}X)}\,(ax)}{z:\forall X.(X{\to}X) \vdash_{\text{ML}} z:(\varphi{\to}\varphi){\to}(\varphi{\to}\varphi)}\,(\forall\mathcal{E}) \quad \dfrac{\dfrac{}{z:\forall X.(X{\to}X) \vdash_{\text{ML}} z:\forall X.(X{\to}X)}\,(ax)}{z:\forall X.(X{\to}X) \vdash_{\text{ML}} z:\varphi{\to}\varphi}\,(\forall\mathcal{E})}{z:\forall X.(X{\to}X) \vdash_{\text{ML}} z\,z:\varphi{\to}\varphi}\,(\to\mathcal{E})}{\vdash_{\text{ML}} \mathit{let}\ z = \lambda y.y\ \texttt{in}\ z\,z:\varphi{\to}\varphi}\,(\mathit{let})$$

Although ML admits less polymorphism than System F does, it has the advantage of being very practical - not only is type assignment in ML decidable (in contrast to System F), but it has a principal type property. Milner presents an algorithm (called $\mathcal{W}$) that takes as input a pair of context and term (Γ, M) and returns a pair of substitution and type (S, A),

representing the most general typing for the term (if one exists) using an instantiation of the context Γ.

The formal results concerning the algorithm depend on the following definition (essentially from [24]):

Definition 4.2.5 (Generic Instance)**.** *A type scheme $\overline{A} = \forall\overrightarrow{X_i}.A$ has a* generic instance *$\overline{B} = \forall\overrightarrow{Y_j}.A'$ if there exist types $\overrightarrow{B_i}$ and atomic types $\overrightarrow{\varphi_j}$ such that*[3] *$\overline{A'} = A[\overrightarrow{B_i/X_i}]\,[\overrightarrow{Y_j/\varphi_j}]$, and $\overrightarrow{\varphi_j \notin \overline{A}}$.*

We write $\overline{A} \succeq \overline{B}$ in this case, read "$\overline{B}$ is a generic instance of $\overline{A}$".

Considering the types as logical formulae, in Natural Deduction terms this definition essentially says that $\overline{A} \succeq \overline{B}$ if and only if we can derive $\overline{B}$ from $\overline{A}$ using a series of $(\forall\mathcal{E})$ steps, followed by a series of $(\forall\mathcal{I})$ steps. For example, according to the definition above, we have $\forall X.(X \to X) \succeq \forall Y.\forall Z.((Y \to Z) \to (Y \to Z))$, which can be understood logically by the following natural deduction derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\forall X.(X \to X) \vdash \forall X.(X \to X)}\,(ax)}{\forall X.(X \to X) \vdash (\varphi \to \varphi') \to (\varphi \to \varphi')}\,(\forall\mathcal{E})}{\forall X.(X \to X) \vdash \forall Z.((\varphi \to Z) \to (\varphi \to Z))}\,(\forall\mathcal{I})}{\forall X.(X \to X) \vdash \forall Y.\forall Z.((Y \to Z) \to (Y \to Z))}\,(\forall\mathcal{I})$$

This notion of derivability gives an intuition as to why when we have $\overline{A} \succeq \overline{B}$ then $\overline{B}$ may be considered 'smaller' or 'less general' than $\overline{A}$.

This is made formal by the following results:

Theorem 4.2.6 (Properties of the algorithm $\mathcal{W}$)**.**

Soundness: *If $\mathcal{W}(\langle\Gamma, M\rangle) = \langle S, A\rangle$ then $(S\,\Gamma) \vdash_{\text{ML}} M : A$.*

Completeness: *If, for a context Γ and term M, there exist S and A such that $(S\,\Gamma) \vdash_{\text{ML}} M : A$ then there exist substitutions S_1 and S_2 and a type B such that $\mathcal{W}(\langle\Gamma, M\rangle) = \langle S_1, B\rangle$ and $(S\,\Gamma) = (S_2 \circ S_1\,\Gamma)$ and $(S_2 \circ S_1\,\overline{B}) \succeq \overline{A}$.*

[3] Note that the following "types" are not well-formed, but A' (for example) forms a part of a well-formed type ($\overline{B}$). The equality we write here just means syntactic equality on these portions of types.

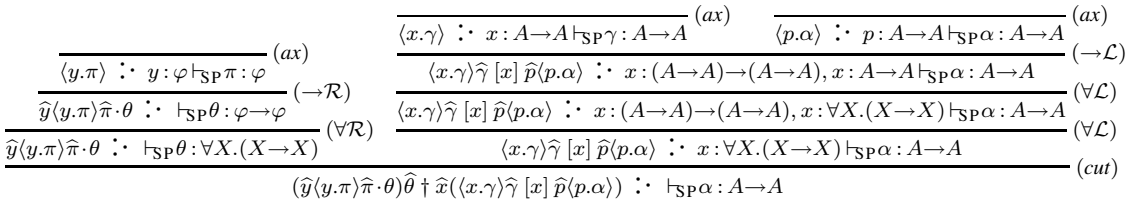

Figure 4.1: Example of shallow-polymorphic type assignment in $\mathcal{X}$

4.2.3 Interlude: Principal Types and Principal Typings

Before we continue with this section, it is important to make clear what we mean by a "principal type property". Wells [104] wrote a paper specifically addressing this point, in which definitions are given for "principal types" and "principal typings". For a type system to have a "principal typing property" there must be an algorithm which, given any term of the syntax, either determines that the term is not typeable at all or else derives *all* of the information used in a typing judgement for the term (other than the term itself), in a *most-general* way. What this information exactly is, and what the notion of 'most general' means depends on the specific calculus and type system. For example, the simply-typed lambda calculus has a principal typing property, for which 'most general' essentially means "can be obtained by applying substitutions and adding extra (redundant) information to the context Γ (weakening)". On the other hand, ML, equipped with the shallow polymorphic type assignment described above, *does not* have a principal typing property [103]. Informally, this essentially is because, given a term M with free variables, it is not possible to determine the most general 'amount' of polymorphism to assume for the types of the free variables. In most cases, the stronger the assumptions made in Γ, the stronger the derived type for M. Instead, ML has a weaker property, which is referred to as a *principal types* property. Essentially, this says that if one *fixes* an initial context of assumptions Γ, as well as a term M, then one can compute the most general pair of substitution S and generic type $\overline{A}$ (if such a pair exist) such that $(S\ \Gamma) \vdash_{\mathrm{ML}} M : \overline{A}$. Since a substitution S cannot affect the quantified (bound) parts of the types in Γ, it can be understood that the initial Γ determines exactly the polymorphic behaviour which will be assumed for the free variables of M. This is what the algorithm $\mathcal{W}$ achieves.

4.3 Universal Shallow Polymorphism for $\mathcal{X}^i$

4.3.1 The Intuitive, Unsound Approach

The key to the use of polymorphism in ML is in the *let* construct, which is interpreted as a substitution both syntactically (according to its reduction rule) and semantically (see [58]). The polymorphism present in the (let)-rule essentially gives a way of typing the substitution about to take place such that multiple occurrences of the name to replace need not all be typed in the same way. The *let*-construct is a necessary extension to the syntax for a shallow polymorphic approach (short of allowing polymorphism to be used directly with abstractions and applications, which leads to System F), since there is nothing in the syntax of the λ-calculus to represent these substitutions.

In the $\mathcal{X}^i$-calculus, there is a construct already present which can be seen to encode substitution. The cut $P\widehat{\alpha} \dagger \widehat{x}Q$ can, by right-evaluation, approximately simulate the substitution of P for the occurrences of x in Q. This observation led to the investigation of a notion of shallow polymorphic type-assignment for the $\mathcal{X}^i$-calculus. Following what seems to be the analogous approach to ML, one adds generic types to the type language, which are allowed to be used for the typing of cuts and axioms (but not the other syntax constructs), and the standard logical rules for $\forall$ (this time for the sequent calculus) are added to the type assignment rules.

Definition 4.3.1 (Naïve Shallow Polymorphic Type Assignment for $\mathcal{X}^i$ [89])**.** *Types A, B and type-schemes $\overline{A}$ are defined as follows:*

$$\begin{aligned} A, B &::= \varphi \mid X \mid (A \to B) \mid (\neg A) \\ \overline{A} &::= \forall X_1.\forall X_2.\ldots.\forall X_n.A \ \ (n \geq 0) \end{aligned}$$

The shallow polymorphic type assignment for $\mathcal{X}^i$ is defined by the following rules (where $\overline{A}$ represents a generic type of Definition 4.2.3):

$$\frac{}{\langle x.\alpha\rangle \ :\cdot\ \Gamma, x:\overline{A} \vdash_{\text{NSP}} \alpha:\overline{A}, \Delta}\ (ax) \qquad \frac{P \ :\cdot\ \Gamma \vdash_{\text{NSP}} \alpha:\overline{A}, \Delta \quad Q \ :\cdot\ \Gamma, x:\overline{A} \vdash_{\text{NSP}} \Delta}{P\widehat{\alpha} \dagger \widehat{x}Q \ :\cdot\ \Gamma \vdash_{\text{NSP}} \Delta}\ (cut)^1$$

$$\frac{P \ :\cdot\ \Gamma \vdash_{\text{NSP}} \alpha:A, \Delta \quad Q \ :\cdot\ \Gamma, x:B \vdash_{\text{NSP}} \Delta}{P\widehat{\alpha}\,[y]\,\widehat{x}Q \ :\cdot\ \Gamma, y:A{\to}B \vdash_{\text{NSP}} \Delta}\ (\to\mathcal{R})^1 \qquad \frac{P \ :\cdot\ \Gamma, x:A \vdash_{\text{NSP}} \alpha:B, \Delta}{\widehat{x}P\widehat{\alpha}{\cdot}\beta \ :\cdot\ \Gamma \vdash_{\text{NSP}} \beta:A{\to}B}\ (\to\mathcal{L})^1$$

$$\frac{P \ :\cdot\ \Gamma \vdash_{\text{NSP}} \alpha:A, \Delta}{x \cdot P\widehat{\alpha} \ :\cdot\ \Gamma, x:\neg A \vdash_{\text{NSP}} \Delta}\ (\neg\mathcal{L})^2 \qquad \frac{P \ :\cdot\ \Gamma, x:A \vdash_{\text{NSP}} \Delta}{\widehat{x}P \cdot \alpha \ :\cdot\ \Gamma \vdash_{\text{NSP}} \alpha:\neg A}\ (\neg\mathcal{R})^3$$

$$\frac{P \ :\cdot\ \Gamma, x:\overline{A}[B/X] \vdash_{\text{NSP}} \Delta}{P \ :\cdot\ \Gamma, x:\forall X.\overline{A} \vdash_{\text{NSP}} \Delta}\ (\forall\mathcal{L}) \qquad \frac{P \ :\cdot\ \Gamma \vdash_{\text{NSP}} \alpha:\overline{A}, \Delta}{P \ :\cdot\ \Gamma \vdash_{\text{NSP}} \alpha:\forall X.\overline{A}[X/\varphi], \Delta}\ (\forall\mathcal{R})^4$$

[1]*: if $x \notin \Gamma$ and $\alpha \notin \Delta$.* [2]*: if $\alpha \notin \Delta$.* [3]*: if $x \notin \Gamma$.* [4]*: if φ does not occur in Γ, Δ.*

Note: despite our conventions on notation, the first three conditions above are explicitly required because, as we will explain in the following discussion, in this type system we need to allow the possibility of multiple type statements for the same connector in a context.

As before, we include a notion of implicit contraction in the above rules (as for the type system presented in Section 3.3), so that if a derivation rule introduces a statement which was already present in the context, it is simply merged.

Notice that generic types are not used in the $(\rightarrow\mathcal{R})$ or $(\rightarrow\mathcal{L})$ rules. This enforces the restriction that the $\forall$-symbol may not appear to the left of an '$\rightarrow$' in a type, and is similar to the way the $(\rightarrow I)$ and $(\rightarrow E)$ rules are treated in ML.

A subtle problem occurs in defining a shallow polymorphic type assignment in this way, which suggests a possible relaxation of Definition 3.3.1 to allow multiple statements in a context with the same subject. The mechanism for taking instances of a type scheme employs the $(\forall\mathcal{L})$ rule, which can be seen to allow instances of the $\forall$ formula to be taken further up in a derivation. However, because the instance $\overline{A}[B/X]$ appears also on the left-hand side of the sequent, and is labelled with the same name (socket), this eliminates the possibility of further instances being taken further up in the same 'branch' of the derivation - the statement $\forall X.\overline{A}$ may not remain in the upper sequent of the rule, since we insist in Definition 3.3.1 that the *subjects* of the statements in a context are distinct. Thinking in terms of the logical proofs however, the subjects of the statements are not a consideration - sequent proofs need not always be annotated (depending on the presentation of the logic) and would certainly allow a use of the $(\forall\mathcal{L})$ rule to include an implicit contraction. For example, in the following proof the formula $\forall X.(X\rightarrow X)$ is used in two $(\forall\mathcal{L})$ rules:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(A\rightarrow A)\vdash(A\rightarrow A)}\,(\mathrm{Ax}) \quad \dfrac{}{(A\rightarrow A)\vdash(A\rightarrow A)}\,(\mathrm{Ax})}{(A\rightarrow A),(A\rightarrow A)\rightarrow(A\rightarrow A)\vdash A\rightarrow A}\,(\rightarrow\mathcal{L})}{\forall X.(X\rightarrow X),(A\rightarrow A)\rightarrow(A\rightarrow A)\vdash A\rightarrow A}\,(\forall\mathcal{L})}{\forall X.(X\rightarrow X)\vdash A\rightarrow A}\,(\forall\mathcal{L})$$

This might correspond to a type derivation in a shallow polymorphic system, (where we use B as a shorthand for the formula $(A\rightarrow A)$) looking like:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x.\alpha\rangle :\cdot x:B \vdash \alpha : B}\,(\mathrm{Ax}) \quad \dfrac{}{\langle y.\beta\rangle :\cdot y:B \vdash \beta : B}\,(\mathrm{Ax})}{\langle x.\alpha\rangle\widehat{\alpha}\,[x]\,\widehat{y}\langle y.\beta\rangle :\cdot x:B, x:B{\to}B \vdash \beta : B}\,(\to\mathcal{L})}{\langle x.\alpha\rangle\widehat{\alpha}\,[x]\,\widehat{y}\langle y.\beta\rangle :\cdot x:\forall X.(X{\to}X), x:B{\to}B \vdash \beta : B}\,(\forall\mathcal{L})}{\langle x.\alpha\rangle\widehat{\alpha}\,[x]\,\widehat{y}\langle y.\beta\rangle :\cdot x:\forall X.(X{\to}X) \vdash \beta : B}\,(\forall\mathcal{L})$$

This is a type derivation we would like to be legal in this system, since we can view this as part of the type derivation for a term analogous to *let* $x = \lambda y.y$ `in` xx, which we wish to be able to type (cf. Figure 4.1). It is possible to work around this problem, by adjusting the set of rules so that instances can be taken implicitly of a quantified formula. In fact, this solution will be employed in the next section, for reasons which will become clear. However, for the moment we explore a more basic solution, which yields a type-system whose underlying derivations are still standard logical proofs.

To deal with the problem of instantiating quantified types in this system, we initially considered relaxing Definition 3.3.1, allowing multiple statements in a context with the same subject. This seems at first glance a risky move, but hopefully the example above has shown that it allows intuitively sound derivations to be constructed. In order to retain soundness, we needed to be careful that whenever a connector is bound, some statements involving the connector do not remain in the context. We therefore insisted that whenever the rules $(\to\mathcal{R})$, $(\to\mathcal{L})$ and (cut) were employed, the connectors mentioned in the top line of the rule (which are bound in the construction of the respective terms) had a unique statement in the rule. This enforces that all the types for a connector disappear from the contexts when the connector is bound. We also insisted that a derivation is not complete unless the subjects of the statements in the *final* sequent are unique (so the relaxation is only usable temporarily within a derivation). As a consequence of these restrictions, if several statements with the same subject (but different types) are used in a derivation, it will be necessary for the $\forall$ rules to be applied until the types of these statements match, and they are contracted into a single statement. Until this takes place, it will be impossible to either bind the connective concerned, or complete the derivation.

This is the type system which was presented in [89], in which a notion of principal contexts (with respect to an initial context) was also defined, in the spirit of the principal types property for ML. As we shall explain next, while this type system seems in many ways analagous to the way polymorphism is introduced to ML, in our more general setting (and particularly in the presence of classical logic), this approach is unsound.

4.3.2 Failure of Subject Reduction

Unfortunately, the 'intuitive' approach outlined in the previous section does not guarantee subject reduction (although it was originally believed to do so [89]. The problem is due to the interaction between the use of implicit (i.e., not represented syntactically in the calculus) polymorphism in the type derivation, and ability to perform left propagation reductions. In particular, since the implicit quantifier rules can occur at any point in a derivation, a left-cut may be propagated 'through' an occurrence of the $(\forall\mathcal{R})$ rule used to type the left-hand subterm. In order to construct a new type derivation for the resulting term, we need to be able to 'relocate' the occurrence of the $(\forall\mathcal{R})$ rule, to be applied further up on the derivation. This is not always possible, because the side-condition of the rule is not always satisfied in this new position. We can make this clearer with an example.

Example 4.3.2 (Failure of Subject Reduction)**.** *Define the term* $P = \widehat{x}(\widehat{y}\langle x.\alpha\rangle\widehat{\alpha}\cdot\gamma)\widehat{\beta}\cdot\gamma$*. This term can be assigned the same contexts as the identity, in the type system presented above:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x.\alpha\rangle \;\vdots\; x:\varphi, y:\varphi \vdash_{\text{NSP}} \alpha:\varphi, \beta:\varphi}\,(ax)}{\widehat{y}\langle x.\alpha\rangle\widehat{\alpha}\cdot\gamma \;\vdots\; x:\varphi \vdash_{\text{NSP}} \beta:\varphi, \gamma:\varphi{\to}\varphi}\,(\to\mathcal{R})}{\widehat{x}(\widehat{y}\langle x.\alpha\rangle\widehat{\alpha}\cdot\gamma)\widehat{\beta}\cdot\gamma \;\vdots\; \emptyset \vdash_{\text{NSP}} \gamma:\varphi{\to}\varphi}\,(\to\mathcal{R})}{\widehat{x}(\widehat{y}\langle x.\alpha\rangle\widehat{\alpha}\cdot\gamma)\widehat{\beta}\cdot\gamma \;\vdots\; \emptyset \vdash_{\text{NSP}} \gamma:\forall X.(X{\to}X)}\,(\forall\mathcal{R})$$

Therefore, if we place this term in a cut which 'applies it to itself' (i.e. in an ML sense, we construct let $z = P$ `in` $z\ z$*), then the resulting term can be typed as follows:*

$$\dfrac{\dfrac{\text{as above}}{P \;\vdots\; \emptyset \vdash_{\text{NSP}} \gamma:\forall X.(X{\to}X)} \qquad \dfrac{\dfrac{}{\langle z.\delta\rangle\widehat{\delta}\,[z]\;\widehat{w}\langle w.\epsilon\rangle \;\vdots\; z:\forall X.(X{\to}X), z:(\varphi'{\to}\varphi'){\to}(\varphi'{\to}\varphi') \vdash_{\text{NSP}} \epsilon:\varphi'{\to}\varphi'}\,(\to\mathcal{L})}{\langle z.\delta\rangle\widehat{\delta}\,[z]\;\widehat{w}\langle w.\epsilon\rangle \;\vdots\; z:\forall X.(X{\to}X) \vdash_{\text{NSP}} \epsilon:\varphi'{\to}\varphi'}\,(\forall\mathcal{L})}{P\widehat{\gamma} \dagger \widehat{z}(\langle z.\delta\rangle\widehat{\delta}\,[z]\;\widehat{w}\langle w.\epsilon\rangle) \;\vdots\; \emptyset \vdash_{\text{NSP}} \epsilon:\varphi'{\to}\varphi'}\,(cut)$$

However, this term can be shown to reduce as follows:

$$\begin{array}{ll}
 & (\widehat{x}(\widehat{y}\langle x.\alpha\rangle\widehat{\alpha}\cdot\gamma)\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}(\langle z.\delta\rangle\widehat{\delta}\,[z]\;\widehat{w}\langle w.\epsilon\rangle) \\
\to & (\widehat{x}((\widehat{y}\langle x.\alpha\rangle\widehat{\alpha}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}(\langle z.\delta\rangle\widehat{\delta}\,[z]\;\widehat{w}\langle w.\epsilon\rangle))\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}(\langle z.\delta\rangle\widehat{\delta}\,[z]\;\widehat{w}\langle w.\epsilon\rangle) & (prop\text{-L}) \\
\to & (\widehat{x}((\widehat{y}\langle x.\alpha\rangle\widehat{\alpha}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}(((\widehat{y}\langle x.\alpha\rangle\widehat{\alpha}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}(z.\delta))\widehat{\delta}\,[z]\;\widehat{w}\langle w.\epsilon\rangle))\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}(\langle z.\delta\rangle\widehat{\delta}\,[z]\;\widehat{w}\langle w.\epsilon\rangle) & (prop\text{-R}) \\
\to & (\widehat{x}((\widehat{y}\langle x.\alpha\rangle\widehat{\alpha}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}((\widehat{y}\langle x.\alpha\rangle\widehat{\alpha}\cdot\delta)\widehat{\delta}\,[z]\;\widehat{w}\langle w.\epsilon\rangle))\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}(\langle z.\delta\rangle\widehat{\delta}\,[z]\;\widehat{w}\langle w.\epsilon\rangle) & (impR) \\
\to & (\widehat{x}((\widehat{y}\langle x.\alpha\rangle\widehat{\alpha}\cdot\delta)\widehat{\delta} \dagger \widehat{y}(\langle x.\alpha\rangle\widehat{\alpha} \dagger \widehat{w}\langle w.\epsilon\rangle))\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}(\langle z.\delta\rangle\widehat{\delta}\,[z]\;\widehat{w}\langle w.\epsilon\rangle) & (imp) \\
\to & (\widehat{x}((\widehat{y}\langle x.\alpha\rangle\widehat{\alpha}\cdot\delta)\widehat{\delta} \dagger \widehat{y}\langle x.\epsilon\rangle)\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}(\langle z.\delta\rangle\widehat{\delta}\,[z]\;\widehat{w}\langle w.\epsilon\rangle) & (cap) \\
\to & (\widehat{x}\langle x.\epsilon\rangle\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}(\langle z.\delta\rangle\widehat{\delta}\,[z]\;\widehat{w}\langle w.\epsilon\rangle) & (prop\text{-R}) \\
\to & (\widehat{x}\langle x.\epsilon\rangle\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}(((\widehat{x}\langle x.\epsilon\rangle\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}(z.\delta))\widehat{\delta}\,[z]\;\widehat{w}\langle w.\epsilon\rangle) & (prop\text{-R}) \\
\to & (\widehat{x}\langle x.\epsilon\rangle\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{z}((\widehat{x}\langle x.\epsilon\rangle\widehat{\beta}\cdot\delta)\widehat{\delta}\,[z]\;\widehat{w}\langle w.\epsilon\rangle) & (cap) \\
\to & (\widehat{x}\langle x.\epsilon\rangle\widehat{\beta}\cdot\delta)\widehat{\delta} \dagger \widehat{x}(\langle x.\epsilon\rangle\widehat{\beta} \dagger \widehat{w}\langle w.\epsilon\rangle) & (imp) \\
\to & (\widehat{x}\langle x.\epsilon\rangle\widehat{\beta}\cdot\delta)\widehat{\delta} \dagger \widehat{x}\langle x.\epsilon\rangle & (prop\text{-L}) \\
\to & \widehat{x}\langle x.\epsilon\rangle\widehat{\beta}\cdot\epsilon & (impR)
\end{array}$$

The resulting term is not typeable in this system. In fact, the problem came right in the first step, when the cut was propagated to the left, through the structure of the term P. In the typing derivation for P, the crucial $(\forall\mathcal{R})$ rule comes right at the very end. But, when propagating a copy of this cut inside the structure of P, in order to maintain the same quantified type for the new cut there must be a similar occurrence of the $(\forall\mathcal{R})$ rule on this copy; i.e., the rule needs to be moved upwards in the derivation with the cut. This is not possible; at this point x is still a free socket in the context, carrying the type φ which is to be generalised by the $(\forall\mathcal{R})$ rule.

In short, the condition on the $(\forall\mathcal{R})$ rule is not necessarily preserved by moving it further up the typing derivation, and so, when cuts are left-propagated 'past' an occurrence of the $(\forall\mathcal{R})$ rule, it is not always possible to rebuild the same quantifier rule in a suitable new position. In general, this means that a type derivation cannot always be reconstructed.

With hindsight, the failure of subject reduction is not that surprising. It is well-known that the standard ML approach to polymorphism is unsound in the presence of various extensions to the language, such as references, exceptions and call/cc [45]. As we will discuss in more detail in Chapter 6, calculi based on classical logic can be closely related to functional calculi extended with control operators, and we believe that (for example), the version of ML with call/cc included could also be encoded into the $\mathcal{X}^i$-calculus. Therefore, the polymorphic type-system presented above must almost inevitably be unsound. However, we believe that the source of the unsoundness is actually much clearer in the sequent calculus setting; it is clear that the attempted left propagation of a cut 'past' an occurrence of $(\forall\mathcal{R})$ in the left-hand typing derivation, is the exact source of the problem. In fact, we can describe the essence of this problem by observing that the presence of the following three aspects will guarantee such an unsoundness:

1. Implicit universal quantification.
2. Call-by-value reductions (not necessarily *only* these reductions, but their inclusion in the calculus).
3. Ability to express/encode classical structural rules (e.g., contraction) manipulating statements on the right of a typing sequent.

Our counter-example depends on the presence of these three features. Implicit quantification allows reduction to 'ignore' the quantifier steps which are violated in the example. Left-propagation of a cut which could be right-propagated (i.e., a call-by-value reduction) ensures that such a violation cannot be 'fixed' in the reduct (i.e., there is in general no way of typing the reduct by, for example, resorting to non-quantified types). Finally, (implicit)

right-contraction in the typing is used to cause the failure of the side-condition on the $(\forall\mathcal{R})$ after left-propagation is performed.

It is interesting to note that examples exist in the literature of proposed calculi and type systems which include each possible pair of *two* out of the three ingredients for unsoundness described. The ML calculus has implicit universal quantification, and call-by-value reductions, but no classical features such as right-contraction in the type system. Parigot's presentation of the $\lambda\mu$-calculus in [67] includes implicit universal quantification, and the ability to (indirectly) express right-contraction in the type system, however the reduction rules are essentially restricted to call-by-name reductions. Ong and Stewart's definition of call-by-value $\lambda\mu$-calculus obviously includes call-by-value reductions, and still permits right-contraction to be expressed in the type system, but no rules for polymorphism are included. Therefore, in each of these works, one of the three 'ingredients' described above is missing, and so the unsoundness we are concerned with is avoided.

There are three main approaches described in the literature for dealing with this unsoundness in the context of ML:

1. Introducing a separate class of ('imperative') atomic types [91], which must be used whenever an 'imperative' feature such as call/cc is to be typed, and may not be generalised using the $(\forall\mathcal{R})$ rule. In our setting it is less obvious how to understand this solution, but it amounts essentially to permitting polymorphic types only on cuts where the left-hand subterm satisfies certain properties (we conjecture that these properties amount to the subterm representing a proof valid in *minimal* logic, but this idea is not explored here).

2. Restricting reductions to a call-by-name strategy [57]. It turns out that the problematic cases cannot be reached by call-by-name reductions. The reason for this can be fairly clearly seen in the context of the $\mathcal{X}^i$-calculus and the counter-example presented above; restricting to call-by-name amounts to insisting that cuts be propagated preferentially to the right, and only left-propagated if the socket bound in the right-hand subterm is introduced. In such a situation, any cut which can be typed with a quantified type may also be typed without quantification; this is because the uniqueness of the socket means that the ability to take multiple instances of the quantified type is irrelevant. Therefore, by the time a cut is left-propagated, we may depend essentially on the subject reduction property of the simple type system.

3. Restricting the form of let-bound terms to bind only values [105] (i.e., only allow terms of the form *let* $x = V$ $\mathtt{in}$ M. Again, the soundness of this approach can be seen clearly in our setting; restricting to values here amounts to restricting the left-hand subterm of cuts $P\widehat{\alpha}\dagger\widehat{x}Q$ to the cases where P introduces α. In such a

system, no left-propagation reductions can ever take place, and so the problematic scenario is avoided.

Unfortunately, given that our original aim was to eventually define a type system in which both existential and universal quantification could be employed, none of the above solutions above seem very desirable. We will explain why this is so for each in turn:

1. This approach breaks the logical foundation of the type system, and, while practical in the ML setting, it is not clear how it would be adapted to deal with existential quantification, and whether a useful system would result.

2. Unfortunately, just as a call-by-name reduction strategy is required to make implicit universal quantification safe, a call-by-value strategy would be needed to ensure subject reduction for a system with similar existential polymorphism. Thus, no reduction strategy would work for a system with both kinds of polymorphism.

3. Similarly, in order to ensure that subject reduction held for a system with both kinds of quantifiers, one would need to restrict the system to allow both kinds of quantifiers in the typing of cuts $P\widehat{\alpha} * \widehat{x}Q$ only in the case when both P introduces α and Q introduces x. Ensuring this condition was met and preserved by reduction would result in a system with almost no useful polymorphism - these cuts can be typed just as well in the simple (non-polymorphic) type system.

By examining the problematic cases in more detail, we were able to come up with a fourth solution (which is in fact, a generalisation of the restriction to values, above). This is, to restrict the points in a derivation where polymorphic generalisation (i.e., the $(\forall\mathcal{R})$ rule in the previous type system) may be employed. In order to avoid the unsoundness described, we only allow generalisation of a statement immediately when it is introduced into the derivation. For example, when the $(\rightarrow\mathcal{R})$ rule is applied, the type for the exhibited plug may be generalised, but if it is not, then it cannot be later on in the derivation. The advantages of our solution are that it imposes fewer restrictions on the type system than the restriction to values (more terms are typeable), and that it does not eliminate in principle the possibility of a useful extended system based on both existential and universal quantification.

The observation we have gained from the sequent calculus setting is that the unsoundness of the naïve system is directly caused by the left-propagation of cuts $P\widehat{\alpha} * \widehat{x}Q$ past occurrences of the $(\forall\mathcal{R})$ rule in the typing derivation for P. We note that this can only only happen because, in general, it is allowed for such occurrences to exist in positions

far devoid from the points in the derivation (and term) where occurrences of α are exhibited. Since it is these points which the left-propagation of the cut reaches, the cut can of course pass over occurrences of the $(\forall\mathcal{R})$ rule on the way. The third solution listed above ([105]) can be understood then as removing the possibility of such 'gaps' between the occurrences of α and the occurrences of $(\forall\mathcal{R})$ applied to the type of α; by insisting on the strong requirement that P introduces α, i.e., that there is exactly one occurrence of α in P, and that it is at the top-level, any $(\forall\mathcal{R})$ rules to be used in typing the cut must also occur at this top level. However, we observe that it would suffice to guarantee the weaker property, that there are no 'gaps' between each occurrence of α and the polymorphic rules applied to the type for the occurrence; this guarantees that a cut 'seeking out' the occurrences of α need never cross over such rules.

Consider the following $\mathcal{X}^i$-term, for example (where the subterm P is left unspecified):

$$((\widehat{x}\langle x.\alpha\rangle\widehat{\alpha}\cdot\beta)\widehat{\epsilon}\,[i]\,\widehat{j}(\widehat{x}\langle x.\alpha\rangle\widehat{\alpha}\cdot\beta))\widehat{\beta}\dagger\widehat{z}P$$

The left-hand subterm of the cut contains two copies of the identity function $\widehat{x}\langle x.\alpha\rangle\widehat{\alpha}\cdot\beta$, both of which exhibit occurrences of the output β (the other names within the terms are also identical, but since these are bound it is only for comparison). The two 'copies' of the identity are independent of one another (the surrounding import does not bind any plugs/sockets in the subterms, and acts as a 'dummy context' for this example). Since each copy can be given the polymorphic type $\forall X.(X\rightarrow X)$, it seems reasonable for the cut to employ this type, also. Furthermore, since the $(\forall\mathcal{R})$ rule applications needed to derive the type $\forall X.(X\rightarrow X)$ can be located at each of the points where β is exhibited, there is no need to risk the possibility of the cut 'crossing' these rules by left-propagation. Essentially, if the polymorphic generalisation steps in a derivation can be located at the same syntactic level as the connector whose type they apply to is exhibited, the derivation is safe from the potential unsoundness described above. This notion is tricky to formalise in a type system with rules for manipulating quantifiers independent of the other rules in the type system (e.g., the $(\forall\mathcal{L})$ and $(\forall\mathcal{R})$ rules in the system presented above). However, since we are now proposing that such rules be employed only at the points where the corresponding connectors are introduced, we can instead present a system with the polymorphism steps 'built in' to the other rules. This will be presented next.

4.3.3 An Improved Shallow Polymorphic Type System

As in Chapter 3, we write *typeof* $x\ \Gamma$ and *typeof* $\alpha\ \Delta$ to denote functions which look up the type assigned to the connector by the context, and if none is defined, return a fresh atomic

type. For example, if $\Gamma = \{x : \overline{A}, y : B\}$ then *typeof* $x\ \Gamma = \overline{A}$, while, for $y \neq z \neq x$, *typeof* $z\ \Gamma = \varphi$ for some fresh atomic type φ.

We now extend Definition 4.2.5 to allow the comparison of (right) contexts as follows:

Definition 4.3.3 (Generalised Generic Instance)**.** *A type scheme* $\overline{A} = \forall\overrightarrow{X_i}.A$ *has a* generic instance $\overline{B} = \forall\overrightarrow{Y_j}.A'$ *if there exist types* $\overrightarrow{B_i}$ *and atomic types* $\overrightarrow{\varphi_j}$ *such that we have* $A' = A[\overrightarrow{B_i/X_i}]\,[\overrightarrow{Y_j/\varphi_j}]$*, and* $\overrightarrow{\varphi_j \notin \overline{A}}$.

We write $\overline{A} \succeq \overline{B}$ *in this case, read "*$\overline{B}$ *is a generic instance of* $\overline{A}$*".*

We extend this notion to (right)-contexts Δ_1, Δ_2 *as follows:* $\Delta_1 \succeq \Delta_2 \quad \Leftrightarrow \quad (\alpha \in \Delta_1 \Rightarrow \alpha \in \Delta_2\ \&\ (\textit{typeof}\ \alpha\ \Delta_1) \succeq (\textit{typeof}\ \alpha\ \Delta_2)$.

Similarly to Definition 4.2.5, we can give a logical understanding of this definition. Previously we described a relationship with the $\forall$-fragment of natural deduction, but the same holds true for the sequent calculus: we have $\overline{A} \succeq \overline{B}$ if and only if the sequent $\overline{A} \vdash \overline{B}$ is derivable using only $\forall$-fragment of the logic (i.e., the inference rules $(ax), (\forall\mathcal{L}), (\forall\mathcal{R})$). To take the same example as previously, $\forall X.(X \to X) \succeq \forall Y.\forall Z.((Y \to Z) \to (Y \to Z))$ holds, which can be understood by the following sequent calculus derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(\varphi \to \varphi') \to (\varphi \to \varphi') \vdash (\varphi \to \varphi') \to (\varphi \to \varphi')}\,(ax)}{\forall X.(X \to X) \vdash (\varphi \to \varphi') \to (\varphi \to \varphi')}\,(\forall\mathcal{L})}{\forall X.(X \to X) \vdash \forall Z.((\varphi \to Z) \to (\varphi \to Z))}\,(\forall\mathcal{R})}{\forall X.(X \to X) \vdash \forall Y.\forall Z.((Y \to Z) \to (Y \to Z))}\,(\forall\mathcal{R})$$

More generally, for contexts Δ_1 and Δ_2, if $\Delta_1 \succeq \Delta_2$ then the sequent $\Delta_1 \vdash \Delta_2$ is derivable in the logic (so again, in logical terms, Δ_2 is 'smaller' or 'less general' than Δ_1).

It is also useful to have an explicit notation for a 'closure' relation on types, which characterises the behaviour of the $\forall\mathcal{R}$ rule. This rule can be used to replace types with more general (larger, in the $\succeq$ relation) forms, provided this is sound with respect to the context in which it is used. Thus this relation depends not only on the types which are changed, but also on the types present in the rest of the context (cf. the condition on the $\forall\mathcal{R}$ rule). We introduce a relation on types which coincides with any number of valid $\forall\mathcal{R}$ steps being applied to the same statement $\alpha : \overline{A}$ say, in a context $\langle\Gamma; \Delta\rangle$.

Definition 4.3.4 (Closures and fresh instances)**.** *1. For any type schemes* $\overline{A}, \overline{B}$ *and context* $\langle\Gamma; \Delta\rangle$*, we say* $\overline{A}$ closes to $\overline{B}$ in $\langle\Gamma; \Delta\rangle$*, and write* $\overline{A} \lhd_{\langle\Gamma;\Delta\rangle} \overline{B}$*, if and only if there exist* $\overrightarrow{X_i}$ *and* $\overrightarrow{\varphi_i}$ *such that* $\overline{B} = \forall\overrightarrow{X_i}.\overline{A}[\overrightarrow{X_i/\varphi_i}]$*, where* $\varphi_i \notin \langle\Gamma; \Delta\rangle$ *and* $\varphi_i \notin \overline{B}$.

2. *For any generic type* $\overline{A} = \overrightarrow{\forall X_i}.A$*, we define freshInst*$(\overline{A}) = A[\overrightarrow{\varphi_i/X_i}]$ *where the* $\overrightarrow{\varphi_i}$ *are fresh atomic types.*

We have the following results for these definitions:

Proposition 4.3.5. *1.* $\preceq$ *is a preorder on type schemes.*

2. *For any contexts* $\langle\Gamma;\Delta\rangle$, $\lhd_{\langle\Gamma;\Delta\rangle}$ *is a partial order on type schemes.*

3. *For any contexts* $\langle\Gamma;\Delta\rangle$ *and type schemes* $\overline{A},\overline{B},\overline{C}$*, if* $\overline{A}\succeq\overline{B}$ *and* $\overline{B} \lhd_{\langle\Gamma;\Delta\rangle} \overline{C}$ *and* $\overline{A} \in \langle\Gamma;\Delta\rangle$*, then* $\overline{A}\succeq\overline{C}$*.*

4. *For any generic types* $\overline{A},\overline{B}$ *and substitution* S*, if* $\overline{A}\succeq\overline{B}$ *then* $(S\ \overline{A})\succeq(S\ \overline{B})$*.*

5. *For any generic types* $\overline{A},\overline{B}$*, context* $\langle\Gamma;\Delta\rangle$ *and substitution* S*, if* $\overline{A} \lhd_{\langle\Gamma;\Delta\rangle} \overline{B}$ *then there exists a substitution* S' *such that dom*$(S') \subseteq (atoms(\overline{A})\backslash atoms(\langle\Gamma;\Delta\rangle))$ *and* $(S'\ \overline{B}) = \overline{B}$ *and* $(S{\circ}S'\ \overline{A}) \lhd_{\langle(S\ \Gamma);(S\ \Delta)\rangle} (S\ \overline{B})$*.*

6. *For any Curry type* A*, type schemes* $\overline{A}$ *and* $\overline{B}$*, and contexts* $\langle\Gamma;\Delta\rangle$*, if* $A \lhd_{\langle\Gamma;\Delta\rangle} \overline{A}$ *and* $\overline{A}\succeq\overline{B}$ *then there is a substitution* S *with dom*$(S) \subseteq (atoms(A)\backslash atoms(\langle\Gamma;\Delta\rangle))$ *and* $(S\ A) \lhd_{\langle\Gamma;\Delta\rangle} \overline{B}$*.*

7. *For any type* $\overline{A} = \overrightarrow{\forall X_i}.A$ *and Curry type* B*, if* $A' =$ *freshInst*$(\overline{A}) = A[\overrightarrow{\varphi_i/X_i}]$ *and* $\overline{A}\succeq B$ *then there exists a substitution* S *such that dom*$(S) = \{\overrightarrow{\varphi_i}\}$ *and* $(S\ A') = B$*.*

Proof. See Proof A.2.1 in Section A.2. □

Definition 4.3.6 (Improved Shallow Polymorphic Type Assignment for $\mathcal{X}^i$)**.** *The (sound) shallow polymorphic type assignment for* $\mathcal{X}^i$ *is defined by the following rules (where* $\overline{A}$ *represents a generic type of Definition 4.2.3):*

$$\frac{}{\langle x.\alpha\rangle \ :\cdot\ \Gamma, x:\overline{A}\vdash_{\mathrm{SP}}\alpha:\overline{B},\Delta}\ (ax)^1 \qquad \frac{P\ :\cdot\ \Gamma, x:A\vdash_{\mathrm{SP}}\alpha:B,\Delta}{\widehat{x}P\widehat{\alpha}\cdot\beta\ :\cdot\ \Gamma\vdash_{\mathrm{SP}}\beta:\overline{C},\Delta}\ (\rightarrow\mathcal{R})^2$$

$$\frac{P\ :\cdot\ \Gamma\vdash_{\mathrm{SP}}\alpha:A,\Delta \quad Q\ :\cdot\ \Gamma, y:B\vdash_{\mathrm{SP}}\Delta}{P\widehat{\alpha}\,[x]\,\widehat{y}Q\ :\cdot\ \Gamma, x:\overline{C}\vdash_{\mathrm{SP}}\Delta}\ (\rightarrow\mathcal{L})^3 \qquad \frac{P\ :\cdot\ \Gamma, x:A\vdash_{\mathrm{SP}}\Delta}{\widehat{x}P\cdot\alpha\ :\cdot\ \Gamma\vdash_{\mathrm{SP}}\alpha:\overline{B},\Delta}\ (\neg\mathcal{R})^4$$

$$\frac{P\ :\cdot\ \Gamma\vdash_{\mathrm{SP}}\alpha:A,\Delta}{x\cdot P\widehat{\alpha}\ :\cdot\ \Gamma, x:\overline{B}\vdash_{\mathrm{SP}}\Delta}\ (\neg\mathcal{L})^5 \qquad \frac{P\ :\cdot\ \Gamma\vdash_{\mathrm{SP}}\alpha:\overline{A},\Delta \quad Q\ :\cdot\ \Gamma, x:\overline{A}\vdash_{\mathrm{SP}}\Delta}{P\widehat{\alpha}\dagger\widehat{x}Q\ :\cdot\ \Gamma\vdash_{\mathrm{SP}}\Delta}\ (cut)$$

$^1\ \overline{A}\succeq\overline{B}$ $\quad^2\ (A{\rightarrow}B) \lhd_{\langle\Gamma;\Delta\rangle} \overline{C}$ $\quad^3\ \overline{C}\succeq(A{\rightarrow}B)$ $\quad^4\ (\neg A) \lhd_{\langle\Gamma;\Delta\rangle} \overline{B}$ $\quad^5\ \overline{B}\succeq(\neg A)$

In comparison with the previous (unsound) proposal, this type system can be seen as a restriction in which the (now implicit) uses of quantifier rules, which could previously occur at any apparently valid point in a type derivation, are now restricted to be applied in

precise positions. In fact, all such quantifier rules are implicitly applied immediately after the statement which they affect is introduced into the context. For example, an occurrence of the $(\forall\mathcal{L})$ rule in the naïve type system, which bound a statement originally introduced by an occurrence of the $(\rightarrow\mathcal{L})$ rule, is (in the new type system) implicitly included in the new version of the $(\rightarrow\mathcal{L})$ rule, by allowing the type for x in the premise to be a generic instance of the type for x in the conclusion of the rule. This essentially permits any number of implicit applications of the $(\forall\mathcal{L})$ rule here.

We can show the following properties for this type system:

Proposition 4.3.7 (Basic properties)**.** *1. For all substitutions S, if $P :\cdot \Gamma \vdash_{\text{SP}} \Delta$ then $P :\cdot (S\,\Gamma) \vdash_{\text{SP}} (S\,\Delta)$.*

2. (Weakening) If $P :\cdot \Gamma \vdash_{\text{SP}} \Delta$, and $\langle \Gamma \cup \Gamma' ; \Delta \cup \Delta' \rangle$ is a well-formed context, then then $P :\cdot \Gamma \cup \Gamma' \vdash_{\text{SP}} \Delta \cup \Delta'$.

3. (Strengthening) If $P :\cdot \Gamma \cup \Gamma' \vdash_{\text{SP}} \Delta \cup \Delta'$, with no sockets x occurring both in Γ' and in fs(P), and similarly no plugs α occurring in both Δ' and fp(P), then $P :\cdot \Gamma \vdash_{\text{SP}} \Delta$.

4. If $P :\cdot \Gamma, x : \overline{B} \vdash_{\text{SP}} \Delta$ and $\overline{A} \succeq \overline{B}$ then $P :\cdot (\Gamma \backslash x), x : \overline{A} \vdash_{\text{SP}} \Delta$.

5. If $P :\cdot \Gamma \vdash_{\text{SP}} \Delta, \alpha : \overline{A}$ and $\overline{A} \succeq \overline{B}$ then $P :\cdot \Gamma \vdash_{\text{SP}} (\Delta \backslash \alpha), \alpha : \overline{B}$.

6. If $P :\cdot \Gamma \vdash_{\text{SP}} \Delta$ and $\Delta \succeq \Delta'$ then $P :\cdot \Gamma \vdash_{\text{SP}} \Delta'$.

Proof. See Proof A.2.2 in Section A.2. □

The new type system is a proper restriction of the old one, which can be understood by adding back the explicit quantifier rules in the naïve type system wherever the $\succeq$ and $\lhd_{\langle \Gamma ; \Delta \rangle}$ relations are employed in the improved type system. We omit the rather-lengthly details here, since we do not depend on this result. However, in brief, in the case of the (ax) rule one employs an (ax) rule followed by a (possibly empty) sequence of $(\forall\mathcal{L})$ rules, followed by a (possibly empty) sequence of $(\forall\mathcal{R})$ rules. In all other cases which employ $\succeq$, a (possibly empty) sequence of $(\forall\mathcal{L})$ rules is added. In all cases which employ $\lhd_{\langle \Gamma ; \Delta \rangle}$, a (possibly empty) sequence of $(\forall\mathcal{R})$ rules is added.

In order to deal succinctly with the more-complex inference rules of the improved type system in the following proofs, we employ the following straightforward lemma:

Lemma 4.3.8 (Generation Lemma)**.** *1. $\langle x.\alpha \rangle :\cdot \Gamma \vdash_{\text{SP}} \Delta$ if and only if $\Gamma = \Gamma', x : \overline{A}$ and $\Delta = \alpha : \overline{B}, \Delta'$ with $\overline{A} \succeq \overline{B}$.*

2. $\widehat{x}P\widehat{\alpha}\cdot\beta :\cdot \Gamma \vdash_{\text{SP}} \Delta$ *if and only if* $x \notin \Gamma$ *and* $\alpha \notin \Delta$ *and* $\Delta = \Delta', \beta : \overline{C}$ *and there exist* A,B *such that* $A \rightarrow B \lhd_{\langle \Gamma;\Delta' \rangle} \overline{C}$ *and* $P :\cdot \Gamma, x : A \vdash_{\text{SP}} \alpha : B, \Delta'$.

3. $P\widehat{\alpha}\,[x]\,\widehat{y}Q :\cdot \Gamma \vdash_{\text{SP}} \Delta$ *if and only if* $\alpha \notin \Delta$ *and* $y \notin \Gamma$ *and* $\Gamma = \Gamma', x : \overline{C}$ *and there exist* A,B *such that* $\overline{C} \succeq A \rightarrow B$ *and* $P :\cdot \Gamma' \vdash_{\text{SP}} \alpha : A, \Delta$ *and* $Q :\cdot \Gamma', y : B \vdash_{\text{SP}} \Delta$.

4. $\widehat{x}P \cdot \alpha :\cdot \Gamma \vdash_{\text{SP}} \Delta$ *if and only if* $x \notin \Gamma$ *and* $\Delta = \Delta', \alpha : \overline{B}$ *and there exists* A *such that* $\neg A \lhd_{\langle \Gamma;\Delta' \rangle} \overline{B}$ *and* $P :\cdot \Gamma, x : A \vdash_{\text{SP}} \Delta$.

5. $x \cdot P\widehat{\alpha} :\cdot \Gamma \vdash_{\text{SP}} \Delta$ *if and only if* $\alpha \notin \Delta$ *and* $\Gamma = \Gamma', x : \overline{B}$ *and there exists* A *such that* $\overline{B} \succeq \neg A$ *and* $P :\cdot \Gamma' \vdash_{\text{SP}} \alpha : A, \Delta$.

6. $P\widehat{\alpha}\,\dagger\,\widehat{x}Q :\cdot \Gamma \vdash_{\text{SP}} \Delta$ *if and only if* $\alpha \notin \Delta$ *and* $x \notin \Gamma$ *and there exists* $\overline{A}$ *such that* $P :\cdot \Gamma \vdash_{\text{SP}} \alpha : \overline{A}, \Delta$ *and* $Q :\cdot \Gamma, x : \overline{A} \vdash_{\text{SP}} \Delta$.

Proof. Each case follows from the fact that each syntactic construct can be typed by a unique typing rule, imposing exactly the conditions described. □

We can now show that this new type system amends the unsoundness of the previous one.

Theorem 4.3.9 (Witness Reduction for Improved Type Assignment)**.** *1. If both of the following hold:*

$$P :\cdot \Gamma \vdash_{\text{SP}} \alpha : \overline{A}, \Delta \tag{4.1}$$

$$Q :\cdot \Gamma, x : \overline{A} \vdash_{\text{SP}} \Delta \tag{4.2}$$

then we have:

(a) $Q\{P\widehat{\alpha} \leftrightarrow x\} :\cdot \Gamma \vdash_{\text{SP}} \Delta$

(b) $P\{\alpha \leftrightarrow \widehat{x}Q\} :\cdot \Gamma \vdash_{\text{SP}} \Delta$

2. *If* $P :\cdot \Gamma \vdash_{\text{SP}} \Delta$ *and* $P \rightarrow Q$ *then* $Q :\cdot \Gamma \vdash_{\text{SP}} \Delta$.

Proof. See Proof A.2.3 in Section A.2. □

Using our previous observation concerning the fact that *let* and a cut can both explicitly represent a substitution, we define an encoding of the language of ML into $\mathcal{X}^i$.

Definition 4.3.10 (Encoding ML in $\mathcal{X}^i$)**.**

$$\begin{aligned}
\llbracket x \rrbracket_\alpha^{\mathrm{ML}} &= \langle x.\alpha \rangle \\
\llbracket \lambda x.M \rrbracket_\alpha^{\mathrm{ML}} &= \widehat{x} \llbracket M \rrbracket_\beta^{\mathrm{ML}} \widehat{\beta} \cdot \alpha \\
\llbracket MN \rrbracket_\alpha^{\mathrm{ML}} &= \llbracket M \rrbracket_\beta^{\mathrm{ML}} \widehat{\beta} \dagger \widehat{y} (\llbracket N \rrbracket_\gamma^{\mathrm{ML}} \widehat{\gamma} \, [y] \, \widehat{z} \langle z.\alpha \rangle) \\
\llbracket \mathit{let}\ x = M\ \mathtt{in}\ N \rrbracket_\alpha^{\mathrm{ML}} &= \llbracket M \rrbracket_\beta^{\mathrm{ML}} \widehat{\beta} \dagger \widehat{x} \llbracket N \rrbracket_\alpha^{\mathrm{ML}}
\end{aligned}$$

where y, z, β, γ are fresh connectors.

This is an extension of the encoding of λ-calculus given for the $\mathcal{X}$-calculus [98]. In all cases there is exactly one occurrence of the plug α in the resulting $\mathcal{X}^i$-term, and this is the only free plug.

Lemma 4.3.11 (Cuts simulate substitutions)**.**

1. *For all ML-terms M, N, $\llbracket N \rrbracket_\alpha^{\mathrm{ML}} \{ \llbracket M \rrbracket_\beta^{\mathrm{ML}} \widehat{\beta} \leftrightarrow x \} \to \llbracket (N[M/x]) \rrbracket_\alpha^{\mathrm{ML}}$.*
2. *For all ML-terms M, N, $\llbracket M \rrbracket_\beta^{\mathrm{ML}} \widehat{\beta} \dagger \widehat{x} \llbracket N \rrbracket_\alpha^{\mathrm{ML}} \to \llbracket (N[M/x]) \rrbracket_\alpha^{\mathrm{ML}}$.*

Proof. 1. By straightforward induction on the structure of the term N.

2. From the previous part.

□

The fact that such a cut behaves like the substitution of the original system relies on the fact that β occurs only once in the left-hand subterm of the cut. If an arbitrary $\mathcal{X}^i$-term were to appear here in which β occurred many times, the cut might be activated to the left (via the rule $(\mathit{act}\text{-L})$), and copies of the right-hand term made during propagation; then the behaviour might be quite different.

Returning to our encoding of ML, we have the following result.

Theorem 4.3.12 (Simulation of ML)**.** *For all* ML*-terms M, N, if $M \to_{ML} N$ then $\llbracket M \rrbracket_\alpha^{\mathrm{ML}} \to \llbracket N \rrbracket_\alpha^{\mathrm{ML}}$.*

Proof. Examining definition 4.2.2, there are two cases to consider.

$(M \equiv (\lambda x.M_1\ M_2))$ Then $N \equiv M_1[M_2/x]$. Applying Definition 4.3.10, we can see that:

$$\begin{aligned}
\llbracket(\lambda x.M_1)\ M_2\rrbracket^{\mathrm{ML}}_{\beta} &= \llbracket\lambda x.M_1\rrbracket^{\mathrm{ML}}_{\delta}\widehat{\delta} \dagger \widehat{y}(\llbracket M_2\rrbracket^{\mathrm{ML}}_{\epsilon}\widehat{\epsilon}\,[y]\ \widehat{z}\langle z.\beta\rangle) \\
&= (\widehat{x}\llbracket M_1\rrbracket^{\mathrm{ML}}_{\phi}\widehat{\phi}\cdot\delta)\widehat{\delta} \dagger \widehat{y}(\llbracket M_2\rrbracket^{\mathrm{ML}}_{\epsilon}\widehat{\epsilon}\,[y]\ \widehat{z}\langle z.\beta\rangle) \\
&\rightarrow \llbracket M_2\rrbracket^{\mathrm{ML}}_{\epsilon}\widehat{\epsilon} \dagger \widehat{x}(\llbracket M_1\rrbracket^{\mathrm{ML}}_{\phi}\widehat{\phi} \dagger \widehat{z}\langle z.\beta\rangle) \\
&\rightarrow \llbracket M_2\rrbracket^{\mathrm{ML}}_{\epsilon}\widehat{\epsilon} \dagger \widehat{x}(\llbracket M_1\rrbracket^{\mathrm{ML}}_{\phi}[\beta/\phi]) \\
&= \llbracket M_2\rrbracket^{\mathrm{ML}}_{\epsilon}\widehat{\epsilon} \dagger \widehat{x}\llbracket M_1\rrbracket^{\mathrm{ML}}_{\beta}
\end{aligned}$$

This case is completed by Lemma 4.3.11.

$(M \equiv \mathit{let}\ x = M_2\ \mathtt{in}\ M_1)$ Again, $N \equiv M_1[M_2/x]$. The result follows immediately from Lemma 4.3.11, noting that

$$\llbracket \mathit{let}\ x = M_2\ \mathtt{in}\ M_1\rrbracket^{\mathrm{ML}}_{\beta} = \llbracket M_2\rrbracket^{\mathrm{ML}}_{\epsilon}\widehat{\epsilon} \dagger \widehat{x}\llbracket M_1\rrbracket^{\mathrm{ML}}_{\beta}$$

□

We can show that our type system is at least as flexible as the restricted type system for ML which we have generalised:

Proposition 4.3.13 (Preservation of Typings)**.** *For all* ML*-terms* M*, if* $\Gamma \vdash_{\mathrm{ML}} M : \overline{A}$ *in the type system in which polymorphism is restricted to let-terms which bind values [105], then* $\llbracket M\rrbracket^{\mathrm{ML}}_{\beta} :\cdot\ \Gamma \vdash_{\mathrm{SP}} \beta : \overline{A}$*.*

It is natural to ask whether our generalisation is useful in the original context of ML. For example, can we define a type system for ML based on the observations of this chapter which allows more typeable terms than [105], and is still sound? We can answer this question in the affirmative; if we define a type system via our encoding into the $\mathcal{X}^i$-calculus (i.e., we encode ML-terms and then type their encodings), we obtain a more permissive system. A simple example of this extra flexibility can be seen as follows:

Example 4.3.14 (Enhanced type assignment for ML)**.** *Consider the ML-term let* $x = (\mathit{let}\ y = \lambda z.z\ \mathtt{in}\ y)\ \mathtt{in}\ x$*. This term reduces to the identity* $\lambda z.z$*, and it would be nice if this could be reflected by its assignable types. Furthermore, since there are no free variables in the term (and no imperative features, of course), it seems as though it must be safe to do so. However, the value restriction [105] does not permit the outermost let-construct to employ a polymorphic type, since let* $y = \lambda z.z\ \mathtt{in}\ y$ *is not a value. Considering the encoding into* $\mathcal{X}^i$*; we obtain (using the plug* α *as output of the whole term)* $((\widehat{z}\langle z.\beta\rangle\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{y}\langle y.\delta\rangle)\widehat{\delta} \dagger \widehat{x}\langle x.\alpha\rangle$*. The polymorphic type derivable for the term*

$\widehat{z}\langle z.\beta\rangle\widehat{\beta}\cdot\gamma$ can be 'carried through' each of the cuts, and in particular, when the subterm $\langle y.\delta\rangle$ is typed, the polymorphic type can be assigned to δ immediately, as is required by the system. The first part of such a typing derivation follows (the outermost cut is typed analogously to the one shown):

$$\dfrac{\dfrac{\dfrac{}{\langle z.\beta\rangle \;\vdots\; z:\varphi\vdash_{\mathrm{SP}}\beta:\varphi}\,(ax)}{\widehat{z}\langle z.\beta\rangle\widehat{\beta}\cdot\gamma \;\vdots\; \emptyset\vdash_{\mathrm{SP}}\gamma:\forall X.(X\to X)}\,(\to\mathcal{R}) \qquad \dfrac{}{\langle y.\delta\rangle \;\vdots\; y:\forall X.(X\to X)\vdash_{\mathrm{SP}}\delta:\forall X.(X\to X)}\,(ax)}{(\widehat{z}\langle z.\beta\rangle\widehat{\beta}\cdot\gamma)\widehat{\gamma}\dagger\widehat{y}\langle y.\delta\rangle \;\vdots\; \emptyset\vdash_{\mathrm{SP}}\delta:\forall X.(X\to X)}\,(cut)$$

Therefore, the application of our ideas in the ML setting yields a more-permissive type system than that proposed by Wright [105].

4.3.4 Principal Contexts

It is well known that a notion of principal types for ML terms exists (as presented by Milner), with respect to an initial basis Γ. This result is shown through the definition of the algorithm $\mathcal{W}$, which takes as input an ML-term and initial basis Γ, and can be used to compute the most general pair of substitution S and generic type $\overline{A}$ such that $(S\,\Gamma)\vdash_{\mathrm{ML}} M:\overline{A}$.

In the case of Milner's algorithm $\mathcal{W}$, the types returned are not quantified, but in showing the completeness of the algorithm the $\forall$-closure of the type (see Definition 4.3.15 below) is taken. The closure can be seen to convert a type into its most general form, and so it can be argued that the principal type should be defined after this closure is taken. This is the idea we follow here; we will generalise the types of our outputs as much as possible, in our definition of a principal context.

We can define principal contexts in our shallow polymorphic version of $\mathcal{X}^i$, with respect to a given initial left-context Γ which gives types to the free sockets in a term. We define an algorithm, based loosely on the $\mathcal{W}$ algorithm of [24], which takes as input an $\mathcal{X}^i$-term P and a left-context Γ, and either fails (in which case P is not typeable) or else produces a pair of substitution S and right-context Δ, representing the least substitution and strongest right-context possible such that $P :\cdot (S\,\Gamma)\vdash_{\mathrm{SP}}\Delta$. Before we are able to define this algorithm, we need to define a number of 'helper' operations.

Firstly, we require an operation to take the 'strongest' closure of a generic type $\overline{A}$ in a context $\langle\Gamma;\Delta\rangle$; essentially this implicitly applies the $(\forall\mathcal{R})$ to the appropriate statement as many times as is possible. Viewed otherwise, the operation computes the 'largest' (in the $\succeq$ relation) generic type $\overline{B}$, such that $\overline{A}\lhd_{\langle\Gamma;\Delta\rangle}\overline{B}$.

Definition 4.3.15 ($\forall$-closure)**.** *The* $\forall$-closure *of type* $\overline{A}$ *with respect to a context* $\langle \Gamma; \Delta \rangle$*, is defined by:* $\forall$*-closure* $\overline{A}$ $\langle \Gamma; \Delta \rangle = \forall X_1 \ldots \forall X_n.(\overline{A}[\overrightarrow{X_i/\varphi_i}])$ *where* $\varphi_1, \ldots, \varphi_n$ *are exactly the atomic types occurring in* $\overline{A}$ *but not in* $\langle \Gamma; \Delta \rangle$.

The process of $\forall$-closure may be seen as taking the 'largest' possible form of a type, in terms of the ordering imposed by $\preceq$. We can show that this operation does indeed compute the 'largest' possible type, by the following result:

Proposition 4.3.16 ($\forall$-closure is the most general closure)**.** *If* $\overline{B} = \forall$*-closure* $\overline{A}$ $\langle \Gamma; \Delta \rangle$ *then:*

1. $\overline{A} \lhd_{\langle \Gamma;\Delta \rangle} \overline{B}$.
2. *If* $\overline{A} \lhd_{\langle \Gamma;\Delta \rangle} \overline{C}$ *then* $\overline{B} \succeq \overline{C}$.
3. *For all substitutions* S, $(S\ \overline{B}) \succeq \forall$*-closure* $(S\ \overline{A})$ $\langle (S\ \Gamma); (S\ \Delta) \rangle$.
4. *If* $\Delta \succeq \Delta'$ *then* $\overline{B} \succeq \forall$*-closure* $\overline{A}$ $\langle \Gamma; \Delta' \rangle$.

Proof. See Proof A.2.4 in Section A.2. □

In our amended type system, whenever a statement is introduced into a right-context it may be 'closed' to a stronger type (with more $\forall$ quantification). Furthermore, this is the only point in the derivation at which these kinds of generalisations may be applied to the statement. For our type-inference algorithm to compute the most general right-context possible, it will use the operation of $\forall$*-closure* whenever such closures are permitted by the rules, in order to obtain the strongest possible type so far. For example, if we were to run our algorithm on the term $\widehat{x}\langle x.\alpha\rangle\widehat{\alpha}\cdot\beta$, we would expect it to generate a type such as $(\varphi \rightarrow \varphi)$ for β but then to also quantify (close) it to the most-general possible type $\forall X.(X \rightarrow X)$.

This approach seems in line with the presentation of our type inference rules; we are employing them in the most-general way possible. However, it leads to a new problem when the contraction of multiple occurrences of a plug β in a term takes place. In general, different quantified types get computed by the algorithm for the different occurrences of a plug β, and at some stage these need to be 'merged' into just one type that works in all positions. In a simple type system, without quantified types, one usually applies Robinson's unification algorithm to perform this 'merging'. However, we need to deal with the fact that quantifiers will, in general, occur in the types. Furthermore, we wish the resulting type to itself be quantified as much as possible.

This leads to a desire for an operation which, given two generic types $\overline{A}$ and $\overline{B}$, computes a third generic type $\overline{C}$ which is the 'most general' type which can be used in place of both $\overline{A}$ and $\overline{B}$. This has parallels with unification; indeed we would expect that if both $\overline{A}$ and $\overline{B}$ contain no quantifiers, then it would perform exactly the operation of unification. On the other hand, if $\overline{A}$ and $\overline{B}$ contained no atomic types, it would seem reasonable that the operation should compute the 'biggest' (in the $\succeq$ sense) generic type which is a generic instance of both $\overline{A}$ and $\overline{B}$. In general, we seek the 'biggest' generic type $\overline{C}$ and minimal substitution S such that both $(S\,\overline{A})\succeq\overline{C}$ and $(S\,\overline{B})\succeq\overline{C}$. Informally, we seek a most general solution in S and $\overline{C}$ to the problem:

$$(S\,\overline{A})\succeq\overline{C}\ \&\ (S\,\overline{B})\succeq\overline{C}$$

We define an algorithm, which we call 'generic unification', in order to compute this 'most general solution'. In order to do so, we need to introduce operations to modify the domains of substitutions. This is because, during the algorithm, fresh instances of the generic types will be taken, and the substitutions subsequently defined will (in general) act on the fresh atomic types introduced. However, these types were not present in the original generic types, and so the resulting substitution would not be the most general one; it might perform the minimal operations on $\overline{A}$ and $\overline{B}$ but *also* perform other operations which are redundant from the point of view of the initial problem.

In order to overcome these difficulties, we define two new operations on substitutions. Firstly, we define the *restriction* of a substitution S to a set of atomic types Φ, which is written $(S \cap \Phi)$ and is itself a substitution which acts on elements of Φ exactly as S does, and on all other atomic types as the identity substitution.

As a shorthand, we also define a complementary operation $(S \cap (\mathit{dom}(S)\backslash\Phi))$ (i.e. restricting a substitution to everything *but* the set Φ), which we write as $(S\backslash\Phi)$ and read as "S without Φ".

We give formal definitions as follows:

Definition 4.3.17 (Restriction of a substitution)**.** *For any substitution S and set of atomic types Φ, the* restriction of S to Φ*, written $(S \cap \Phi)$ is defined by:*

$$(S \cap \Phi) = \{(\varphi \mapsto A) | (\varphi \mapsto A) \in S\ \&\ \varphi \in S\}$$

We also define the shorthand:

$$(S\backslash\Phi) = (S \cap (\mathit{dom}(S)\backslash\Phi)) = \{(\varphi \mapsto A) | (\varphi \mapsto A) \in S\ \&\ \varphi \notin S\}$$

In order to reason formally about the effect of these operations later on, we will require a number of properties about their definitions.

Lemma 4.3.18 (Range and domain)**.**

1. *For any substitutions S_1, S_2 if dom$(S_1)\cap$range$(S_2) = \emptyset$ and dom$(S_2)\cap$range$(S_1) = \emptyset$ then $(S_2 \circ S_1) = (S_1 \circ S_2)$.*

2. *If $S_2 \circ S_1 = S_4 \circ S_3$ and dom$(S_2) \cap$ range$(S_1) = \emptyset$ and dom$(S_2) \cap$ dom$(S_3) = \emptyset$ and dom$(S_2) \cap$ range$(S_3) = \emptyset$, then there exists a substitution S_5 such that $S_1 = S_5 \circ S_3$.*

3. *For any substitution S, type scheme $\overline{A}$ and atomic type φ, if $\varphi \in (S\ \overline{A})$ then either:*

   (a) $\varphi \in$ atoms$(\overline{A})$ and $\varphi \notin$ dom(S), or,

   (b) $\varphi \notin$ atoms$(\overline{A})$ and there exists $\varphi' \in$ atoms$(\overline{A})$ with $\varphi \in$ atoms$(S\ \varphi')$.

4. *For any binding renaming $\overrightarrow{[X_i/\varphi_i]}$, and any type scheme $\overline{A}$ and atomic type φ, if $\varphi \in$ atoms$(\overline{A}\overrightarrow{[X_i/\varphi_i]})$ then $\varphi \in$ atoms$(\overline{A})$ and $\varphi \notin \{\overrightarrow{\varphi_i}\}$.*

Lemma 4.3.19 (Restrictions)**.** 1. *If atoms$(\overline{A}) \subseteq \{\overrightarrow{\varphi_i}\}$ then $(S \cap \{\overrightarrow{\varphi_i}\}\ \overline{A}) = \overline{A}$.*

2. *For any substitution S, type scheme $\overline{A}$ and set of atomic types $\{\vec{\varphi}\}$, if atoms$(\overline{A}) \cap \{\vec{\varphi}\} = \emptyset$ then $(S\ \overline{A}) = ((S \backslash \{\vec{\varphi}\})\ \overline{A})$.*

3. *For any substitution S and set of atomic types $\{\vec{\varphi}\}$, if dom$(S) \subseteq \{\vec{\varphi}\}$ then $S \backslash \{\vec{\varphi}\} =$ id.*

4. *For any substitutions S_1 and S_2 and set of atomic types $\{\vec{\varphi}\}$, if $\{\vec{\varphi}\} \cap$dom$(S_1) = \emptyset$ then $(S_2 \circ S_1) \backslash \{\vec{\varphi}\} = (S_2 \backslash \{\vec{\varphi}\}) \circ S_1$.*

5. *For any substitution S, generic type $\overline{A}$ and set of atomic types $\{\vec{\varphi}\}$, if $(S\ \overline{A}) = \overline{A}$ then $(S \backslash \{\vec{\varphi}\}\ \overline{A}) = \overline{A}$.*

6. *For any substitutions S,S', if it is the case that for all $\varphi \in$ dom(S'), we have $(S'\ \varphi) = (S\ \varphi)$, then it holds that $S' = S \cap$ dom(S').*

7. *For any substitution S and set of atomic types $\{\vec{\varphi}\}$, we have:*

$$S = ((S \cap \{\vec{\varphi}\}) \circ (S \backslash \{\vec{\varphi}\}) = (S \backslash \{\vec{\varphi}\}) \circ ((S \cap \{\vec{\varphi}\})$$

Armed with these definitions and results, we can now present the definition of generic unification.

Definition 4.3.20 (Generic Unification)**.**

$$
\begin{array}{rcl}
\textit{unifyGen}\ \overline{A}\ \overline{B} & = & (S_r, \overrightarrow{\forall X_i.} C_u[\overrightarrow{X_i/\varphi_i}]\,) \\
& & \textit{where} \\
& & A' = \textit{freshInst}(\overline{A}) \\
& & B' = \textit{freshInst}(\overline{B}) \\
& & S_u = \textit{unify}\ A'\ B' \\
& & C_u = (S_u\ A') \\
& & \{\overrightarrow{\varphi_i}\} = \textit{atoms}(C_u) \backslash (\textit{atoms}(S_u\ \overline{A}) \cup \textit{atoms}(S_u\ \overline{B})) \\
& & S_r = (S_u \cap (\textit{atoms}(\overline{A}) \cup \textit{atoms}(\overline{B})))
\end{array}
$$

Note that this algorithm may fail, in the case where the call *unify* A' B' results in failure. As usual, we do not model the failure case explicitly, but speak of success of failure of the algorithm as a whole.

We can give a formal justification for the definition of the algorithm, using the following results:

Theorem 4.3.21 (Soundness and Completeness of Generic Unification)**.** *For any generic types* $\overline{A}$ *and* $\overline{B}$*:*

1. *(Soundness:) If unifyGen* $\overline{A}\ \overline{B}$ *succeeds, resulting in a pair* $(S_r, \overline{C})$ *then* $(S_r\ \overline{A}) \succeq \overline{C}$ *and* $(S_r\ \overline{B}) \succeq \overline{C}$*.*
2. *(Completeness:) If* S *is a substitution and* $\overline{D}$ *a generic type such that* $(S\ \overline{A}) \succeq \overline{D}$ *and* $(S\ \overline{B}) \succeq \overline{D}$*, then unifyGen* $\overline{A}\ \overline{B}$ *succeeds, resulting in a pair* $(S_r, \overline{C})$*, and there exists a further substitution* S' *such that* $S = S' \circ S_r$ *and* $(S'\ \overline{C}) \succeq \overline{D}$*.*

Proof. See Proof A.2.5 in Section A.2. □

Just as for the simple type assignment system (Definition 3.3.2), we require the generalisation of unification to contexts, we require here the generalisation of generic unification to right-contexts. We choose to omit a concrete definition, but depend on the following properties, which are relatively easy to guarantee given the previous theorem:

Proposition 4.3.22 (Soundness and Completeness of Generic Context Unification)**.** *There exists an algorithm unifyGenContexts which takes two right-contexts* Δ_1 *and* Δ_2 *as arguments, and (if it succeeds) returns a pair of substitution* S_u *and right-context* Δ_u*, satisfying:*

1. *If unifyGenContexts* $\Delta_1\ \Delta_2$ *succeeds, then* $(S_u\ \Delta_1) \succeq \Delta_u$ *and* $(S_u\ \Delta_2) \succeq \Delta_u$*.*

2. *If* S *is a substitution and* Δ *a right-context such that* $(S\ \Delta_1)\succeq\Delta$ *and* $(S\ \Delta_2)\succeq\Delta$*, then unifyGenContexts* $\Delta_1\ \Delta_2$ *succeeds, and there exists a further substitution* S' *such that* $S = S'\circ S_u$ *and* $(S'\ \Delta_u)\succeq\Delta$.

We are now in a position to define our type-inference algorithm.

Definition 4.3.23 (*sppc*)**.** *The procedure* sppc $::\ \langle\mathcal{X}^i,\Gamma\rangle\ \rightarrow\ \langle S,\Delta\rangle$ *is defined in Figure 4.2.*

We can now give our principal contexts result.

Theorem 4.3.24 (Soundness and Completeness of *sppc*)**.** *Given an* $\mathcal{X}^i$*-term* R *and an initial left-context* Γ *such that*

$$\mathit{fs}(R)\subseteq\mathit{dom}(\Gamma) \tag{4.3}$$

we have:

1. *If* sppc (R,Γ) *succeeds and* sppc $(R,\Gamma)=\langle S_R,\Delta_R\rangle$ *then* $R :\cdot (S\ \Gamma)\vdash_{\text{SP}}\Delta_R$.
2. *If there exist* $\langle S,\Delta\rangle$ *such that* $R :\cdot (S\ \Gamma)\vdash_{\text{SP}}\Delta$*, then a call* sppc (R,Γ) *succeeds, and if* sppc $(R,\Gamma)=\langle S_R,\Delta_R\rangle$ *then there exists a further substitution* S' *such that* $S = S'\circ S_R$ *and* $(S'\ \Delta_R)\succeq\Delta$.

Proof. See Proof A.2.6 in Section A.2. □

4.4 Extensions to the Type System

4.4.1 Existential Shallow Polymorphism

Since classical sequent calculus exhibits a natural symmetry between left and right contexts (inputs and outputs, in a computational sense), it is natural to consider the asymmetric notion of (universal) polymorphism presented so far as an incomplete picture. Universal polymorphism allows an output (plug) type to be generalised with quantified variables, and then to be connected to multiple input types, each taking a different instantiation of the variables. What then, if we allow this the opposite way around? It seems natural to consider the generalisation of an *input* type, to be instantiated many times for the multiple *outputs* it is connected with.

In a logical sense, this amounts to the incorporation of *existential* quantification, being the dual notion to universal. As might be hoped, it is straightforward to adapt the earlier work

$$
\begin{array}{rl}
\multicolumn{2}{l}{\mathit{sppc}\,(\langle x.\alpha\rangle,\Gamma) = \langle \mathit{id}, \{\alpha : \overline{A}\}\rangle} \\
\text{where} & \overline{A} = \mathit{typeof}\ x\ \Gamma \\
\multicolumn{2}{l}{\mathit{sppc}\,(\widehat{x}P\widehat{\alpha}\cdot\beta,\Gamma) = \langle S_r, (S_u\ \Delta_P \backslash \alpha \backslash \beta) \cup \{\beta : \overline{D}\}\rangle} \\
\text{where} & \varphi = \mathit{fresh} \\
& \langle S_P, \Delta_P\rangle = \mathit{sppc}\,(P, \Gamma \cup \{x : \varphi\}) \\
& A = (S_P\ \varphi) \\
& B = \mathit{freshInstance\ typeof}\ \alpha\ \Delta_P \\
& \overline{C} = \forall\text{-}\mathit{closure}\ A \rightarrow B\ \langle (S_P\ \Gamma); \Delta_P \backslash \alpha\rangle \\
& \langle S_u, \overline{D}\rangle = \begin{cases} \mathit{unifyGen}\ \overline{C}\ \mathit{typeof}\ \beta\ \Delta_P & \mathit{if}\ \beta \in \Delta_P \\ \langle \mathit{id}, \overline{C}\rangle & \mathit{otherwise} \end{cases} \\
& S_r = (S_u \circ S_P \cap \mathit{atoms}(\Gamma)) \\
\multicolumn{2}{l}{\mathit{sppc}\,(P\widehat{\alpha}\,[y]\,\widehat{x}Q,\Gamma) = \langle S_r, \Delta_c\rangle} \\
\text{where} & \langle S_P, \Delta_P\rangle = \mathit{sppc}\,(P,\Gamma) \\
& \varphi = \mathit{fresh} \\
& \langle S_Q, \Delta_Q\rangle = \mathit{sppc}\,(Q, (S_P\ \Gamma) \cup \{y : \varphi\}) \\
& A = \mathit{freshInstance\ typeof}\ \alpha\ (S_Q\ \Delta_P) \\
& B = (S_Q\ \varphi) \\
& C = \mathit{freshInstance\ typeof}\ x\ (S_Q \circ S_P\ \Gamma) \\
& S_u = \mathit{unify}\ C\ A \rightarrow B \\
& \langle S_c, \Delta_c\rangle = \mathit{unifyGenContexts}\ (S_u \circ S_Q\ \Delta_P \backslash \alpha)\ (S_u\ \Delta_Q) \\
& S_r = (S_c \circ S_u \circ S_Q \circ S_P \cap \mathit{atoms}(\Gamma)) \\
\multicolumn{2}{l}{\mathit{sppc}\,(\widehat{x}P\cdot\beta,\Gamma) = \langle S_r, (S_u\ \Delta_P \backslash \beta) \cup \{\beta : \overline{D}\}\rangle} \\
\text{where} & \varphi = \mathit{fresh} \\
& \langle S_P, \Delta_P\rangle = \mathit{sppc}\,(P, \Gamma \cup \{x : \varphi\}) \\
& A = (S_P\ \varphi) \\
& \overline{C} = \forall\text{-}\mathit{closure}\ \neg A\ \langle (S_P\ \Gamma); \Delta_P\rangle \\
& \langle S_u, \overline{D}\rangle = \begin{cases} \mathit{unifyGen}\ \overline{C}\ \mathit{typeof}\ \beta\ \Delta_P & \mathit{if}\ \beta \in \Delta_P \\ \langle \mathit{id}, \overline{C}\rangle & \mathit{otherwise} \end{cases} \\
& S_r = (S_u \circ S_P \cap \mathit{atoms}(\Gamma)) \\
\multicolumn{2}{l}{\mathit{sppc}\,(y\cdot P\widehat{\alpha},\Gamma) = \langle S_r, (S_u\ \Delta_P)\rangle} \\
\text{where} & \langle S_P, \Delta_P\rangle = \mathit{sppc}\,(P,\Gamma) \\
& A = \mathit{freshInstance\ typeof}\ \alpha\ \Delta_P \\
& C = \mathit{freshInstance\ typeof}\ x\ (S_P\ \Gamma) \\
& S_u = \mathit{unify}\ C\ \neg A \\
& S_r = (S_u \circ S_P \cap \mathit{atoms}(\Gamma)) \\
\multicolumn{2}{l}{\mathit{sppc}\,(P\widehat{\alpha}\,\dagger\,\widehat{x}Q,\Gamma) = \langle S_r, \Delta_c\rangle} \\
\text{where} & \langle S_P, \Delta_P\rangle = \mathit{sppc}\,(P,\Gamma) \\
& \overline{A} = \mathit{typeof}\ \alpha\ \Delta_P \\
& \langle S_Q, \Delta_Q\rangle = \mathit{sppc}\,(Q, (S_P\ \Gamma) \cup \{x : \overline{A}\}) \\
& \langle S_c, \Delta_c\rangle = \mathit{unifyGenContexts}\ (S_Q\ \Delta_P \backslash \alpha)\ \Delta_Q \\
& S_r = (S_c \circ S_Q \circ S_P \cap \mathit{atoms}(\Gamma))
\end{array}
$$

Figure 4.2: Principal Contexts for Shallow Polymorphic system

of this chapter to define a type system allowing (only) existential shallow polymorphism, instead of universal. The essence of the symmetry here can be seen by examination of the quantifier rules from the logic:

$$\frac{\Gamma, \overline{A} \vdash \Delta}{\Gamma, \exists X.\overline{A}[X/\varphi] \vdash \Delta}\,(\exists\mathcal{L})^* \qquad \frac{\Gamma \vdash \overline{A}[B/X], \Delta}{\Gamma \vdash \exists X.\overline{A}, \Delta}\,(\exists\mathcal{R})$$

*: if φ does not occur in Γ, Δ.

All of the work presented in this chapter can be adapted analogously for this alternative quantifier. The notion of *closure* of a type (Definition 4.3.4) is identical for existential quantification. The notion of *generic instance* (Definition 4.2.5) is also the same for the existential quantifier, except that to read $\overline{A} \succeq \overline{B}$ as "$\overline{A}$ is more general than $\overline{B}$" requires the relation to be inverted compared with the definition for $\forall$. The type system which results from these changes is as follows:

Definition 4.4.1 (Existential Shallow Polymorphic Type Assignment for $\mathcal{X}^i$)**.** *The existential shallow polymorphic type assignment for $\mathcal{X}^i$ is defined by the following rules (where $\overline{A}$ represents a shallow existential type):*

$$\frac{}{\langle x.\alpha\rangle \;::\; \Gamma, x:\overline{A} \vdash_{\text{SP}} \alpha : \overline{B}, \Delta}\,(ax)^1 \qquad \frac{P \;::\; \Gamma, x:A \vdash_{\text{SP}} \alpha : B, \Delta}{\widehat{x}P\widehat{\alpha}\cdot\beta \;::\; \Gamma \vdash_{\text{SP}} \beta : \overline{C}, \Delta}\,(\rightarrow\mathcal{R})^2$$

$$\frac{P \;::\; \Gamma \vdash_{\text{SP}} \alpha : A, \Delta \quad Q \;::\; \Gamma, y:B \vdash_{\text{SP}} \Delta}{P\widehat{\alpha}\,[x]\,\widehat{y}Q \;::\; \Gamma, x:\overline{C} \vdash_{\text{SP}} \Delta}\,(\rightarrow\mathcal{L})^3 \qquad \frac{P \;::\; \Gamma, x:A \vdash_{\text{SP}} \Delta}{\widehat{x}P\cdot\beta \;::\; \Gamma \vdash_{\text{SP}} \beta : \overline{C}, \Delta}\,(\neg\mathcal{R})^4$$

$$\frac{P \;::\; \Gamma \vdash_{\text{SP}} \alpha : A, \Delta}{x\cdot P\widehat{\alpha} \;::\; \Gamma, x:\overline{C} \vdash_{\text{SP}} \Delta}\,(\neg\mathcal{L})^5 \qquad \frac{P \;::\; \Gamma \vdash_{\text{SP}} \alpha : \overline{A}, \Delta \quad Q \;::\; \Gamma, x:\overline{A} \vdash_{\text{SP}} \Delta}{P\widehat{\alpha}\,\dagger\,\widehat{x}Q \;::\; \Gamma \vdash_{\text{SP}} \Delta}\,(cut)$$

$^1\ \overline{A} \succeq \overline{B}.$ $^2\ (A \rightarrow B) \succeq \overline{C}.$ $^3\ (A \rightarrow B) \lhd_{\langle\Gamma;\Delta\rangle} \overline{C}.$ $^4\ (\neg A) \succeq \overline{C}.$ $^5\ (\neg A) \lhd_{\langle\Gamma;\Delta\rangle} \overline{C}.$

The resulting type system is sound, although the analogous naïve system would not be (in this case, it is *right* propagation that presents a potential for unsoundness, but this is eliminated above by the same restriction; polymorphic generalisation steps, in the form of the $(\exists\mathcal{L})$ rule, can only be employed at the points at which the appropriate connectors occur).

We can also define a principal typings algorithm, by 'reflecting' the definitions employed in the previous section. In particular, such an algorithm would type a cut by typing first the *right*-hand subterm, and then using the (potentially existentially-quantified) type obtained to help type the left.

It is interesting that existential polymorphism is traditionally understood in the context of *information hiding* [60], i.e., providing a facility to *lose* typing information from a

term, rather than providing extra power in terms of typeability. However, this is a question of paradigm; in a traditional functional setting, based on minimal logic (such as the λ-calculus), the addition of existential quantification does extend the typeable terms, while the addition of universal quantification (along with suitable syntactic constructs such as let-binding) does. This can be readily understood by moving again to the sequent calculus setting; when injecting ML (for example) into $\mathcal{X}^i$, via the translation above, one always obtains a term in which there is exactly one free plug, and exactly one occurrence of the plug. Therefore, the additional power in terms of typeability which existential polymorphism brings, is not applicable, since it caters for the situation when multiple occurrences of the same plug need to be typed in different ways.

To summarise, in the setting of classical sequent calculus, the two variants of polymorphism can be seen exactly as dual to one another; universal polymorphism allowing generalisation of outputs and instantiation at multiple inputs, and vice versa for existential.

4.4.2 Symmetric Shallow Polymorphism?

A possible and natural extension to this work which has not been investigated in depth is the possibility allowing *both* kinds of quantification to be exploited in a shallow polymorphic type system. Since the cuts in the $\mathcal{X}^i$-calculus can simultaneously bind multiple occurrences of *both* inputs and outputs, it seems reasonable that there may be example terms which would be made typeable by such a system.

The main problem envisaged with such a system is decidable type-assignment. In particular, the presented approach to typing a cut seems not to adapt to this setting. In the case of universal polymorphism, a cut is typed by typing the left-hand subterm first, and using the information gained to help type the right. The reverse ordering of subcalls is suitable for a system with existential polymorphism. But with both connectives permitted, there is no obvious approach; it may be that *each* subterm provides some polymorphic behaviour which allows to overcome difficulties in typing the other subterm.

We present here an example to motivate the potential application of such a type-system. For the purposes of this particular example, it is useful to employ the notion of *pairing*, corresponding with logical conjunction ($\wedge$). The sequent calculus rules for this connective can be presented as follows:

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}\,(\wedge\mathcal{R}) \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\,(\wedge\mathcal{L}_1) \qquad \frac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\,(\wedge\mathcal{L}_2)$$

However, rather than extend our types and calculus to accommodate this connective, we can define pairing by $A \wedge B \equiv \neg(A \to \neg B)$. We will treat $A \wedge B$ as a shorthand, in the following discussion. We choose to encode the first two of the three inference rules above (we do not require the third for our example), as follows:

$$\dfrac{\dfrac{\Gamma \vdash A, \Delta \qquad \dfrac{\Gamma \vdash B, \Delta}{\Gamma, \neg B \vdash \Delta}\,(\neg\mathcal{L})}{\Gamma, A \to \neg B \vdash \Delta}\,(\to\mathcal{L})}{\Gamma \vdash \neg(A \to \neg B), \Delta}\,(\neg\mathcal{R}) \qquad \dfrac{\dfrac{\Gamma, A \vdash \Delta}{\Gamma \vdash A \to \neg B, \Delta}\,(\to\mathcal{I})}{\Gamma, \neg(A \to \neg B) \vdash \Delta}\,(\neg\mathcal{L})$$

Correspondingly (by inhabiting these derivations with $\mathcal{X}^i$-terms), when we wish to take two inputs (on x and w, say) and output the corresponding pair (on γ), we use the term $\widehat{y}(\langle x.\alpha\rangle\widehat{\alpha}\ [y]\ \widehat{z}(z \cdot \langle w.\beta\rangle\widehat{\beta})) \cdot \gamma$, which is typeable as follows:

$$\dfrac{\dfrac{\dfrac{}{\langle x.\alpha\rangle :\cdot \Gamma, x : A, w : B \vdash \alpha : A, \Delta}\,(ax) \qquad \dfrac{\dfrac{}{\langle w.\beta\rangle :\cdot \Gamma, w : B \vdash \beta : B, \Delta}\,(ax)}{z \cdot \langle w.\beta\rangle\widehat{\beta} :\cdot \Gamma, w : B, z : \neg B \vdash \Delta}\,(\neg\mathcal{L})}{\langle x.\alpha\rangle\widehat{\alpha}\ [y]\ \widehat{z}(z \cdot \langle w.\beta\rangle\widehat{\beta}) :\cdot \Gamma, x : A, w : B, y : A \to \neg B \vdash \Delta}\,(\to\mathcal{L})}{\widehat{y}(\langle x.\alpha\rangle\widehat{\alpha}\ [y]\ \widehat{z}(z \cdot \langle w.\beta\rangle\widehat{\beta})) \cdot \gamma :\cdot \Gamma, x : A, w : B \vdash \gamma : \neg(A \to \neg B), \Delta}\,(\neg\mathcal{R})$$

On the other hand, corresponding to the rule $(\wedge\mathcal{L}_1)$, when we want a term which takes an input of type $A \wedge B$ (on l, say) and produces an output of type A (on π), we use $l \cdot (\widehat{m}\langle m.\pi\rangle\widehat{\sigma} \cdot \tau)\widehat{\tau}$, which can be typed as follows:

$$\dfrac{\dfrac{\dfrac{}{\langle m.\pi\rangle :\cdot \Gamma, m : A \vdash \pi : A, \sigma : \neg B, \Delta}\,(ax)}{\widehat{m}\langle m.\pi\rangle\widehat{\sigma} \cdot \tau :\cdot \Gamma \vdash \tau : A \to \neg B, \pi : A, \Delta}\,(\to\mathcal{R})}{l \cdot (\widehat{m}\langle m.\pi\rangle\widehat{\sigma} \cdot \tau)\widehat{\tau} :\cdot \Gamma, l : \neg(A \to \neg B) \vdash \pi : A, \Delta}\,(\neg\mathcal{L})$$

Now we are equipped to construct our example. Consider the function which takes an argument and produces a pair of two copies of that argument. This would be expected to have the output type $A \to (A \wedge A)$. The operation of duplicating an input x and producing the pair on γ is (according to the above) represented by:

$$P_1 = \widehat{y}(\langle x.\alpha\rangle\widehat{\alpha}\ [y]\ \widehat{z}(z \cdot \langle x.\beta\rangle\widehat{\beta})) \cdot \gamma$$

and so the function described is $\widehat{x}P_1\widehat{\gamma} \cdot \delta$.

Now consider the pairing of x and w represented by

$$P_2 = \widehat{y}(\langle x.\alpha\rangle\widehat{\alpha}\ [y]\ \widehat{z}(z \cdot \langle w.\beta\rangle\widehat{\beta}))\cdot\gamma$$

and form the function $\widehat{x}P_2\widehat{\gamma}\cdot\delta$. This term has a free input w, of type B, say, and the output type δ is then $A{\rightarrow}(A \wedge B)$. Both of the terms $\widehat{x}P_1\widehat{\gamma}\cdot\delta$ and $\widehat{x}P_2\widehat{\gamma}\cdot\delta$ have (in a sense) an identity function embedded inside; if one considers the function implicitly defined by mapping x to the first projection of γ, this is the identity in both cases. With this in mind, we construct a term which accepts a function of type $A{\rightarrow}(A \wedge B)$ as input (on socket p), extracts this implicit function of type $A{\rightarrow}A$, and then applies the function to itself, leaving the result on output ω. This can be represented by the term:

$$Q = (\widehat{q}(\langle q.\epsilon\rangle\widehat{\epsilon}\ [p]\ \widehat{r}(r \cdot (\widehat{s}\langle s.\eta\rangle\widehat{\sigma}\cdot\lambda)\widehat{\lambda}))\widehat{\eta}\cdot\mu)\widehat{\mu}\ [p]\ \widehat{l}(l \cdot (\widehat{m}\langle m.\pi\rangle\widehat{\sigma}\cdot\tau)\widehat{\tau})$$

in which the two occurrences of p correspond to the two uses of the input, each of which is reconstructed into a function of type $A{\rightarrow}A$ by the surrounding structure. The point of this example is, in order to type the self-application (buried) within this term, we need to reflect in the typing that the function extracted is indeed the identity function, and is not just typeable as $A{\rightarrow}A$, but as $\forall X.(X{\rightarrow}X)$ as usual. However, we now form the term (in which the import binding $\circ$ and o and inputting on n plays no active role, and is present only to allow the two subterms):

$$((\widehat{x}P_1\widehat{\gamma}\cdot\delta)\widehat{\circ}\ [n]\ \widehat{o}(\widehat{x}P_2\widehat{\gamma}\cdot\delta))\widehat{\delta} \dagger \widehat{p}Q$$

In this term, the two occurrences of δ cannot obviously be given a common polymorphic type. Their natural simple types are $C{\rightarrow}(C \wedge C)$ (for any C), and $A{\rightarrow}(A \wedge B)$ (for any A, where B is the type of w). It is possible to unify these types, to give the type $B{\rightarrow}(B{\rightarrow}B)$ to δ, where B is also the type of the free w in P_2, but then the type of δ may not be generalised (B occurs in the context). This means that the self-application within Q cannot be typeable. However, from the structure of Q, we know that the second element of the pair returned by the function on δ is always discarded, and we wish to ignore it in the type-assignment. This is possible with the use of an existential type. We note that the two types $C{\rightarrow}(C \wedge C)$ and $A{\rightarrow}(A \wedge B)$ can both be 'weakened' (applying the $(\exists\mathcal{R})$ rule) to give $\exists Y.(C{\rightarrow}(C \wedge Y))$ and $\exists Y.(A{\rightarrow}(A \wedge Y))$, reflecting the fact that we wish to treat the type in place of Y as irrelevant. But now, A and C are arbitrary, and do not occur in the context, so we can 'close' both types to the common type $\forall X.\exists Y.(X{\rightarrow}(X \wedge Y))$. The presence of the $\forall$-quantifier makes it possible to type the term Q also; the variable X may be instantiated as some type $D{\rightarrow}D$ for the first occurrence of p, and as the corresponding type D for the second. We omit the exact derivation, since it is prohibitively large, but the rough shape of the type derivation for Q is as follows:

$$\dfrac{\dfrac{}{q : D \vdash \epsilon : D}\,(ax) \qquad \dfrac{\dfrac{}{s : D \vdash \eta : D}\,(ax)}{r : D \wedge E \vdash \eta : D}\,(\wedge\mathcal{L}_1)}{\dfrac{p : \forall X.\exists Y.(X \to (X \wedge Y)), q : D \vdash \eta : D}{p : \forall X.\exists Y.(X \to (X \wedge Y)) \vdash \mu : D \to D}\,(\to\mathcal{R})}\,(\to\mathcal{L}) \qquad \dfrac{\dfrac{}{m : D \to D \vdash \pi : D \to D}\,(ax)}{l : (D \to D) \wedge F \vdash \pi : D \to D}\,(\wedge\mathcal{L}_1)$$
$$\dfrac{}{p : \forall X.\exists Y.(X \to (X \wedge Y)) \vdash \pi : D \to D}\,(\to\mathcal{L})$$

This example motivates the use of types involving both quantifiers. The term in question is not typeable in the simple type system, nor in a shallow polymorphic type system using just one of the quantifiers.

Unfortunately, it does not seem that our principal contexts result can be extended to a type system involving both varieties of shallow polymorphism. The reason for this is the lopsided nature of typing cuts: in the $\forall$-based system the left-hand subterm of a cut is typed first, and its most-general output types are *closed* with $\forall$-quantifiers in order to allow the maximum flexibility in typing the right-hand subterm. In the $\exists$-based system, the dual approach is taken, and a cut is typed from right-to-left. However, in the combined system, it seems unclear how this approach would be adapted. It may be that both subterms of a cut are untypeable in the simple (non-polymorphic) type system, but that each contributes some polymorphic behaviour which allows the typing problems in the other to be resolved. Such an algorithm remains future work.

4.5 Summary

We have shown that the problems with ML-style type assignment in the presence of control operators have parallels when defining such a type assignment for calculi based on classical logic. Furthermore, in the case of classical sequent calculus, the source of the potential unsoundness can be readily identified; it is the propagation of cuts past implicit occurrences of polymorphic generalisation rules which causes trouble. With this in mind, we were able to identify a restriction of the naïve system which precisely avoids the problematic cases, while retaining a reasonable degree of flexibility in the type system. Furthermore, the symmetric nature of classical sequent calculus suggested that existential quantification might also play an interesting role in such a type system, and (as we have demonstrated) there seems to be scope for a powerful and symmetric type assignment based on the incorporation of both quantifiers.

We spent considerable effort proving that our formalisation of shallow polymorphism with the universal quantifier is sound, and retains a notion of principal types similar to

that which is known for ML. In the process, we were required to formalise the notion of generic unification, which has a soundness and completeness property of its own.

In the next chapters, we turn to the paradigm of classical natural deduction, and abandon the work presented here for the time being. However, in Chapter 7, we will present work on relating the two paradigms, and show that the results presented in this section can be adapted appropriately for the natural deduction setting.

Chapter 5

A Term Calculus for Classical Natural Deduction

5.1 Overview

In this chapter, we present a programming calculus based on a Curry-Howard Correspondence with a canonical system of classical natural deduction, close to the original formulation due to Gentzen. The calculus is an extension of Parigot's $\lambda\mu$-calculus, which we call $\nu\lambda\mu$-calculus. Our motivation is to achieve a Curry-Howard Correspondence without significantly modifying the original logic. At the same time, we aim for a notion of reduction which generalises those which exist already in the literature for similar calculi. As for the $\mathcal{X}^i$-calculus, we choose to employ both the implication and negation connectives in the logic. From a computational point of view, this involves introducing a separate binder for constructing continuations: we represent continuations as distinct first-class citizens. This will be shown to have advantages in an untyped setting, since the μ-reduction rules of the calculus can treat continuations and functions differently.

We discuss a generalisation of the μ-reduction rules of existing calculi, and give intuitive motivations for the reductions in terms of the type system. We state a principal typing property for the calculus, which generalises the well-known result for the λ-calculus. We compare the $\nu\lambda\mu$-calculus with some existing calculi and show that we can represent and generalise both the $\lambda\mu$ and $\overline{\lambda}\mu\tilde{\mu}$-calculi, preserving typings and reductions.

In the next chapter, we will show that the generalised notion of μ-reduction which we derive here is able to naturally express a notion of *delimited control* (i.e., control behaviour which can be scoped syntactically, rather than applying to complete programs). This result surprisingly arises naturally from the aim of this chapter: to a define canonical term

calculus and notion of reduction for classical natural deduction.

5.2 Background

5.2.1 Natural Deduction for Classical Logic

We give here for reference a fairly-standard set of Gentzen-style natural deduction rules for classical logic with the $\rightarrow$, $\neg$ and $\bot$ connectives.

Definition 5.2.1 (Classical Natural Deduction with $\rightarrow\ \neg\ \bot$)**.** *Formulas (ranged over by A,B) are defined by the following grammar: $A, B\ :=\ \bot \mid \varphi \mid \neg A \mid A{\rightarrow}B$ (in which φ ranges over an infinite set of atomic formulae).*

$$\frac{}{\Gamma, A \vdash A}\,(ax) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A{\rightarrow}B}\,(\rightarrow\mathcal{I}) \qquad \frac{\Gamma \vdash A{\rightarrow}B \quad \Gamma \vdash A}{\Gamma \vdash B}\,(\rightarrow\mathcal{E})$$

$$\frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\,(PC) \qquad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\,(\neg\mathcal{I}) \qquad \frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash \bot}\,(\neg\mathcal{E})$$

The system consists of the usual axiom rule, introduction and elimination rules for the $\neg$ and $\rightarrow$ connectives, and the rule 'proof by contradiction' (sometimes 'reductio ad absurdum'), which makes the logic classical. Gentzen [39] describes that classical natural deduction is obtained by taking the intuitionistic introduction/elimination rules for the connectives, and adding a rule with "special status" (not fitting the introduction/elimination pattern) to make the logic classical. In our case, the (PC) rule fulfils this role. This differs from Gentzen's original choice, in which he adds instead the 'law of excluded middle' (i.e., $A \vee \neg A$ is made an axiom of the logic, for every formula A). We choose this approach partly because we do not treat disjunction as a primitive connective, but also because we follow previous work ([72, 66]). Considering our stated aim of keeping the logic close to its origins, we do not regard this difference as a serious one, since Gentzen himself employs this alternative when proving the equivalence of his calculi with one another, and with those of Hilbert. Note that we omit the standard rule for $(\bot\mathcal{E})$ since it is subsumed by the (PC) rule (when the bound assumption $\neg A$ is introduced by weakening the context Γ). A detailed comparison of various alternative classical logics and corresponding programming calculi can be found in [3].

5.2.2 The $\lambda\mu$-calculus

The $\lambda\mu$-calculus was introduced by Parigot in [66], and has been extensively studied as a calculus relating to classical logic. It is a calculus based essentially on a Curry-Howard correspondence, but the logic with which it corresponds is not presented in the style of Gentzen. The logic is often referred to as 'natural deduction with multiple conclusions', or sometimes even just 'natural deduction', although this is somewhat misleading (as will be discussed below). We recall here the basic definitions.

Definition 5.2.2 ($\lambda\mu$ Syntax)**.** *The syntax of $\lambda\mu$-terms is defined over two distinct infinite sets of variables (one of Roman letters $x, y, \ldots$ and one of Greek letters $\alpha, \beta, \ldots$) by the following syntax*[1]*:*

$$M, N := x \mid \lambda x.M \mid M\ N \mid [\alpha]M \mid \mu\alpha.M$$

The reduction rules of the $\lambda\mu$-calculus rely on an additional special notion of substitution. The syntax $M\langle[\beta](M'\ N)/[\alpha]M'\rangle$ denotes the replacement of all subterms of M of the form $[\alpha]M'$ with the corresponding term $[\beta](M'\ N)$ (this is sometimes referred to as *structural substitution* in the literature). As usual, we assume these substitutions to be capture-free.

Definition 5.2.3 ($\lambda\mu$ Reductions)**.** *The reductions of the $\lambda\mu$-calculus are defined by the following rules:*

$$\begin{array}{lll}
(\beta) & (\lambda x.M)\ N \to M\langle N/x\rangle & \\
(\mu) & (\mu\alpha.M)\ N \to \mu\beta.M\langle[\beta](M'\ N)/[\alpha]M'\rangle & \\
(\mu r) & [\beta]\mu\alpha.M \to M\langle\beta/\alpha\rangle & \\
(\mu\eta) & \mu\alpha.[\alpha]M \to M & \text{if } \alpha \notin M
\end{array}$$

Since we wish to discuss the logic underlying the $\lambda\mu$-calculus, we will also recall the basic type-assignment rules. We omit the rules for quantifiers (e.g., in [67]), since they are not relevant to this chapter, and obscure the correspondence between the logic and the syntax constructs. Type judgements for this calculus are of the form $\Gamma \vdash_{\lambda\mu} M : A \mid \Delta$, in which M is a term, Γ contains type assumptions for the Roman variables, Δ contains type assumptions for the Greek variables, and A is the corresponding type for M.

[1] In the original presentation of [66], the syntax of the λ-calculus was extended only with terms of the form $\mu\alpha.[\beta]M$; i.e. the last two constructs above only occur together. This simplifies the type-assignment for the calculus, at the expense of some logical expressiveness. Various subsequent work (for example [28, 17, 65, 73]) has involved separating these as constructs; we adopt this approach.

Definition 5.2.4 (Type Assignment for $\lambda\mu$)**.**

$$\frac{}{\Gamma, x:A \vdash_{\lambda\mu} x:A \mid \Delta}\,(ax) \qquad \frac{\Gamma, x:A \vdash_{\lambda\mu} M:B \mid \Delta}{\Gamma \vdash_{\lambda\mu} \lambda x.M : A{\to}B \mid \Delta}\,(\to\mathcal{I})$$

$$\frac{\Gamma \vdash_{\lambda\mu} M : A{\to}B \mid \Delta \quad \Gamma \vdash_{\lambda\mu} N : A \mid \Delta}{\Gamma \vdash_{\lambda\mu} M\ N : B \mid \Delta}\,(\to\mathcal{E})$$

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

The logic underlying this type system is *not* an example of a standard natural deduction calculus: the inference rules corresponding to the constructs $\mu\alpha.M$ and $[\alpha]M$ are presented as structural rules, which allow one to manipulate a collection of conclusions, choosing which is the 'current' or 'active' one[2]. These rules do not fit into the introduction/elimination scheme usual for natural deduction rules.[3] The original intention of the natural deduction style (which was to correspond with natural argument as much as possible), and the characterisation of the inference rules in terms of the semantics of the logical connectives (see [72]) no longer applies in an obvious way to this logic. On the other hand, Parigot shows how to relate the logic back to a usual presentation of natural deduction in [66], in which negation occurs within types, as well as implication and bottom ($\bot$). This is achieved by replacing each multiple-conclusion sequent $\Gamma \vdash_{\lambda\mu} M:A \mid \Delta$ by the single-conclusion sequent $\Gamma, \neg\Delta \vdash M:A$, in which $\neg\Delta = \{\alpha:\neg A \mid \alpha:A \in \Delta\}$. Under this transformation, the type-assignment rules become the following:

$$\frac{}{\Gamma, x:A, \neg\Delta \vdash x:A}\,(ax) \qquad \frac{\Gamma, x:A, \neg\Delta \vdash M:B}{\Gamma, \neg\Delta \vdash \lambda x.M : A{\to}B}\,(\to\mathcal{I})$$

$$\frac{\Gamma, \neg\Delta \vdash M : A{\to}B \quad \Gamma, \neg\Delta \vdash N : A}{\Gamma, \neg\Delta \vdash M\ N : B}\,(\to\mathcal{E})$$

$$\frac{\Gamma, \neg\Delta, \alpha:\neg A \vdash M:\bot}{\Gamma, \neg\Delta \vdash \mu\alpha.M : A}\,(\mu) \qquad \frac{\Gamma, \neg\Delta \vdash M:A}{\Gamma, \neg\Delta, \alpha:\neg A \vdash [\alpha]M : \bot}\,(n)$$

The rules (ax), $(\to\mathcal{I})$ and $(\to\mathcal{E})$ are now essentially the familiar rules from the λ-calculus. The (μ) rule can now be seen as a version of the 'proof by contradiction' inference rule,

[2] In fact, these are reminiscent of the *exchange* rule employed in the original sequent calculus [39].

[3] Ong and Stewart [65] present the typing rules for these two constructs as an introduction/elimination pair for the connective $\bot$. However, these rules do not form such a pair in the sense discussed by Gentzen and made explicit by Prawitz [72]. In particular, the rule for μ-binding is not the natural deduction elimination rule for $\bot$, since it binds an assumption.

$$\dfrac{\dfrac{}{\Gamma, \neg\Delta, \alpha : \neg A \vdash \alpha : \neg A}\,(ax) \qquad \Gamma, \neg\Delta \vdash M : A}{\Gamma, \neg\Delta, \alpha : \neg A \vdash [\alpha]M : \bot}\,(\neg\mathcal{E}) \quad \cdots \quad \dfrac{\Gamma, \neg\Delta \vdash M : A}{\Gamma, \neg\Delta, \alpha : \neg A \vdash [\alpha]M : \bot}\,(n)$$

Figure 5.1: Comparison: rule (n) is a restriction of $(\neg\mathcal{E})$

while the (n) rule is related to the $(\neg\mathcal{E})$ in which the left-hand premise has been restricted to an axiom (but using a Greek variable), as is illustrated in Figure 5.1. In this way, the $\lambda\mu$-calculus can be seen to have a Curry-Howard correspondence with a restricted version of the natural deduction system of Definition 5.2.1. The precise restrictions in place are as follows:

1. Assumptions are divided into two 'classes' (corresponding to the two classes of variables in $\lambda\mu$): this is the reason for the two contexts Γ and Δ in the typing judgements. In this discussion, we will refer to these as 'usual' (Γ) and 'special' (Δ) assumptions.

2. The (PC) rule is restricted to bind only 'special' assumptions in its premise (this corresponds to the μ-binding of Greek variables).

3. The $(\neg\mathcal{E})$ rule is restricted to allow only axioms to occur as the first (major) premise, and these axioms may only feature 'special' assumptions (this corresponds to only allowing Greek variables to occur in the position of α in $[\alpha]M$).

4. 'Special' assumptions may not be used in any other way (Greek variables do not occur in the position of Roman variables).

5. The $(\neg\mathcal{I})$ rule is removed (no syntax construct is present to 'inhabit' this rule).

These restrictions do not seem very intuitive from the point of view of the logic. For example, in the $(\neg\mathcal{E})$ rule, it should be possible for an arbitrary proof of the conclusion $\neg A$ to occur in the position of the major premise. It is natural to consider the effect of these restrictions on the expressiveness of $\lambda\mu$ as a term assignment for classical logic in general. In the presentation of $\lambda\mu$ given in [67], $\bot$ is not given a full treatment as a type (in fact, $\bot$ is not explicitly mentioned in the definition of types, although is referred to later on). Parigot writes "…[we use] the following special interpretation of naming for $\bot$: for α a μ-variable, $\bot^\alpha$ is not mentioned (in fact one could have a special variable φ for $\bot$)". This implicit treatment of what is essentially a $(\bot\mathcal{E})$ step appears to make the Curry-Howard correspondence with the full logic incomplete (although this is open to some debate, depending on whether one identifies $\bot$ with an empty stoup in the judgements). Ariola and

Herbelin argue in [3] that the $\lambda\mu$ calculus corresponds with 'minimal classical logic'[4]. They define an extension of $\lambda\mu$, adding a special syntax construct $[tp]M$, where *tp* acts as a 'continuation constant'. In logical terms, the new construct corresponds with an explicit $(\bot\mathcal{E})$, and they then show that full classical provability is achieved. It seems surprising that the addition of the $(\bot\mathcal{E})$ rule to the logic provides any additional strength in terms of provability, since this rule is (in a standard natural deduction setting) subsumed by the (PC) rule, which is already inhabited by the μ-binding construct. This apparent inconsistency stems from the fact that the presentation of the type system of $\lambda\mu$ (cf. Definition 5.2.4) is quite different from a usual natural deduction presentation. In terms of *provability*, the apparent 'gap' in the original system could be resolved simply by interpreting an empty stoup as a stoup with type $\bot$ inside. In fact, this is essentially the approach taken in [73, 17].

Although completeness from a provability perspective is a definite requirement in order to consider a calculus to represent the full computational content of classical logic, we argue that it is not sufficient. Since we interpret "proofs as programs", it is the *proofs* that give us our computational objects, and the proof reductions which essentially specify the possible computational behaviour (c.f. Section 1.1). Therefore, in order to speak about a Curry-Howard correspondence with full classical logic, as well as ensuring that all valid formulas are provable we should be concerned that all 'interesting' proofs of these formulas are represented.

We consider here each of the five restrictions identified above, and make appropriate additions and alterations to the $\lambda\mu$-calculus so that they can be lifted, with the aim of restoring a Curry-Howard correspondence with a Gentzen-style natural deduction system. In this way, we obtain a calculus still much in the spirit of the $\lambda\mu$-calculus, but with a richer and more expressive syntax. Starting then from the version of $\lambda\mu$ defined above, we make the following changes (corresponding to each of the five points previously identified):

1. The two classes of variables are collapsed into a single set of (Roman) variables.
2. μ-binders now bind the usual term variables: terms of the form $\mu x.M$ are allowed.
3. Terms of the form $[M]N$ are allowed (i.e. there is no restriction on the term M).
4. Greek variables no longer occur in the syntax at all (due to point 1.).

[4]Minimal classical logic is defined in [3] as minimal logic extended with Pierce's law $((A{\to}B){\to}A){\to}A$ but without the rule $(\bot\mathcal{E})$ (which is not derivable in this logic). It is between minimal and classical logic in strength, but distinct from intuitionistic logic (which is minimal logic plus $(\bot\mathcal{E})$).

5. A syntax construct inhabiting the $(\neg\mathcal{I})$ rule is added, which involves a third kind of binder. We write these new terms as $\nu x.M$ (in which x is bound)[5].

In this way, we obtain a syntax which exactly inhabits the natural deduction system of interest. However, we have yet to define the reduction rules for this syntax, and so develop it into a full programming calculus. This is the aim of the next section.

5.3 Syntax and Type Assignment

Definition 5.3.1 (Syntax)**.** *The syntax of the $\nu\lambda\mu$-calculus is defined over the set of variables $x, y, z, \ldots$ as follows:*

$$\begin{array}{rcll} M, N & := & x & \textit{variable} \\ & | & \lambda x.M & \textit{function abstraction} \\ & | & M\ N & \textit{function application} \\ & | & \mu x.M & \textit{context consumer} \\ & | & \nu x.M & \textit{continuation abstraction} \\ & | & [M]N & \textit{continuation application} \end{array}$$

We write fvM for the set of free variables occurring in M, defined as usual.

The descriptions attached to the syntax constructs will be explained below. The typeable fragment of the syntax gives a term representation for the classical natural deduction system of Definition 5.2.1. This can be seen by the following type assignment system for the calculus.

Definition 5.3.2 (Type assignment for $\nu\lambda\mu$-calculus)**.** *Types (ranged over by A,B) are defined over an infinite set of atomic types $\varphi_1, \varphi_2, \ldots$ by the following syntax: $A, B := \bot \mid \varphi \mid \neg A \mid A \rightarrow B$.*
In the sequel, Γ is a finite set of statements $\{x : A, y : B, \ldots\}$ in which no variable may occur more than once. We write $\Gamma, x : A$ to mean $\Gamma \cup \{x : A\}$, and assume that this restriction is always respected (i.e., it is always either the case that x is not mentioned in Γ, or that $x : A \in \Gamma$).
We write $\Gamma \vdash M : A$ to mean that there is a derivation using the rules below with this

[5] We are aware that the notation ν for a binder is already overloaded in the literature, but have not found a satisfactory alternative, and hope this does not cause confusion.

statement as the last line. The type assignment rules are as follows:

$$\frac{}{\Gamma, x : A \vdash x : A}\,(ax) \qquad \frac{\Gamma, x : \neg A \vdash M : \bot}{\Gamma \vdash \mu x.M : A}\,(PC)$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A{\to}B}\,(\to\mathcal{I}) \qquad \frac{\Gamma \vdash M : A{\to}B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\,(\to\mathcal{E})$$

$$\frac{\Gamma, x : A \vdash M : \bot}{\Gamma \vdash \nu x.M : \neg A}\,(\neg\mathcal{I}) \qquad \frac{\Gamma \vdash M : \neg A \quad \Gamma \vdash N : A}{\Gamma \vdash [M]N : \bot}\,(\neg\mathcal{E})$$

As usual, implication is regarded as a function type: $A{\to}B$ denotes a function from A to B. The bottom ($\bot$) type is given to a term to denote that the term does not produce an 'answer' (sometimes such terms are referred to as 'silent'). In particular, a term of type $\bot$ may never occur (typeably) on the left of any kind of application. Negation is interpreted as the type for continuations; the type $\neg A$ represents a continuation, which expects an argument of type A, but does not return anything. One could compare this with the type $A{\to}\bot$.

Example 5.3.3. *A common dual notion to the usual Modus Ponens ($\to\mathcal{E}$) in logic, known as Modus Tollens, is given as follows: 'from $A{\to}B$ and $\neg B$ one can deduce $\neg A$'. This notion can be inhabited by syntax: given a term M of type $A{\to}B$, and a term N of type $\neg B$, we can form the term $\nu x.[N](M\ x)$, which is typeable as follows:*

$$\dfrac{\dfrac{\Gamma, x : A \vdash N : \neg B \qquad \dfrac{\Gamma, x : A \vdash M : A{\to}B \qquad \dfrac{}{\Gamma, x : A \vdash x : A}\,(ax)}{\Gamma, x : A \vdash M\ x : B}\,(\to\mathcal{E})}{\Gamma, x : A \vdash [N](M\ x) : \bot}\,(\neg\mathcal{E})}{\Gamma \vdash \nu x.[N](M\ x) : \neg A}\,(\neg\mathcal{I})$$

In fact, this is the simplest term inhabiting this type. What does it mean in terms of computational behaviour? M is a function from A to B, while N is a continuation with a 'hole' of type B. The result is a continuation which, given a term x of type A, feeds it through the function M and then passes the result (of type B) on to N. Thus we have a kind of 'dual' to function application: the term above combines a function with its continuation, and produces a new continuation as a result, without yet specifying any argument to the function. This is suggestive of a symmetrical view of functions: both their input and their output can be interacted with. However, note that no reduction has taken place within the term M: to directly combine the continuation N with the body of the function M requires the features of the calculus relating to classical logic, which we shortly describe.

5.3.1 Principal Typings

We define a principal typing algorithm for the type system defined above, which generalises the standard result for the λ-calculus. This is defined using unification and substitution, as presented in Definitions 3.3.7 and 3.3.8.

Definition 5.3.4 (Principal Typing Algorithm for $\nu\lambda\mu$)**.** *The algorithm pt takes a $\nu\lambda\mu$[term as argument, and returns a pair of context Γ and type A. It is defined recursively on the structure of the term as follows:*

$$
\begin{array}{rcl}
\mathit{pt}\,(x) & = & \langle\{x:\varphi\},\varphi\rangle \\
& \mathit{where} & \varphi = \mathit{fresh} \\
\mathit{pt}\,(\lambda x.M) & = & \langle\Gamma\backslash x, A{\rightarrow}B\rangle \\
& \mathit{where} & \langle\Gamma,B\rangle = \mathit{pt}\,(M) \\
& & A = \mathit{typeof}\,x\;\Gamma \\
\mathit{pt}\,(M\;N) & = & \langle(S_2{\circ}S_1\;\Gamma_1){\cup}(S_2{\circ}S_1\;\Gamma_2),(S_2{\circ}S_1\;A)\rangle \\
& \mathit{where} & \langle\Gamma_1,A\rangle = \mathit{pt}\,(M) \\
& & \langle\Gamma_2,B\rangle = \mathit{pt}\,(N) \\
& & \varphi = \mathit{fresh} \\
& & S_1 = \mathit{unifyContexts}\;\Gamma_1\;\Gamma_2 \\
& & S_2 = \mathit{unify}\;(S_1\;A)\;(S_1\;B{\rightarrow}\varphi) \\
\mathit{pt}\,(\nu x.M) & = & \mathit{if}\;B = \bot\;\mathit{then}\;\langle\Gamma\backslash x,\neg A\rangle \\
& \mathit{where} & \langle\Gamma,B\rangle = \mathit{pt}\,(M) \\
& & A = \mathit{typeof}\,x\;\Gamma \\
\mathit{pt}\,([M]N) & = & \langle(S_2{\circ}S_1\;\Gamma_1){\cup}(S_2{\circ}S_1\;\Gamma_2),\bot\rangle \\
& \mathit{where} & \langle\Gamma_1,A\rangle = \mathit{pt}\,(M) \\
& & \langle\Gamma_2,B\rangle = \mathit{pt}\,(N) \\
& & \varphi = \mathit{fresh} \\
& & S_1 = \mathit{unifyContexts}\;\Gamma_1\;\Gamma_2 \\
& & S_2 = \mathit{unify}\;(S_1\;A)\;(S_1\;\neg B) \\
\mathit{pt}\,(\mu x.M) & = & \mathit{if}\;B = \bot\;\mathit{then}\;\langle\Gamma\backslash x,(S\;\varphi)\rangle \\
& \mathit{where} & \langle\Gamma,B\rangle = \mathit{pt}\,(M) \\
& & A = \mathit{typeof}\,x\;\Gamma \\
& & \varphi = \mathit{fresh} \\
& & S = \mathit{unify}\;A\;\neg\varphi
\end{array}
$$

Since the algorithm is defined recursively on the structure of the term, it is clear that it always terminates. The algorithm may fail if any of the required unifications fail, or if the condition $B = \bot$ is not met in the cases for terms $\nu x.M$ and $\mu x.M$. Again, we choose

to abstract away the details of dealing with such failures, and simply assume that if any failures are encountered, the whole algorithm fails. From here onward, whenever we make an assertion $pt\,(M) = \langle \Gamma, A \rangle$ we make the implicit assumption that the algorithm succeeded.

The following results show that the algorithm above does indeed compute principal typings:

Proposition 5.3.5 (Principal typings for $\nu\lambda\mu$)**.** *For any $\nu\lambda\mu$ term M, we have the following two properties:*

Soundness *If $pt\,(M) = \langle \Gamma, A \rangle$ then $\Gamma \vdash M : A$.*

Completeness *If there exist Γ', B such that $\Gamma' \vdash M : B$ then $pt\,(M)$ succeeds, and there exists a substitution S such that $(S\,\Gamma) \subseteq \Gamma'$ and $(S\,A) = B$ where $\langle \Gamma, A \rangle = pt\,(M)$.*

Proof. Both by induction on the structure of the term M. The arguments follow similar lines to the proofs of Theorem 3.3.12 (and are slightly simpler, due to the single contexts in the type derivations), and are omitted here. □

5.4 Reduction Rules

We write $\rightarrow_{\nu\lambda\mu}$ to denote the reflexive, transitive, compatible closure of the reduction rules which we define for the $\nu\lambda\mu$-calculus. However, so long as it is not confusing, we will usually drop the subscript and just write $\rightarrow$ for this relation.

5.4.1 β-Reductions

The reduction rules (λ) and (ν) below are the standard logical rules for the $\rightarrow$ and $\neg$ connectives[6], in which $M\langle N/x \rangle$ denotes the *implicit* substitution of the term N in place of all occurrences of the variable x in the term M. As usual, we assume all substitutions in this chapter to be capture-free.

$$(\lambda)\ (\lambda x.M)\ N \rightarrow M\langle N/x \rangle$$
$$(\nu)\quad [\nu x.M]N \rightarrow M\langle N/x \rangle$$

[6]The rule (λ) is of course the usual (β) rule of the λ-calculus; we rename it here only because *both* of the rules below are similar to (β).

We now wish to present an intuitive reading for those syntax constructs (Definition 5.3.1) not inherited from the λ-calculus. The μ-bound terms provide the control behaviour of the calculus, and we will deal with them later on in this chapter, since they require substantial discussion. A ν-bound term provides an explicit representation for constructing a continuation: a term which expects an input but does not produce a meaningful output. Terms of the form $[M]N$ represent the application of the continuation M to the argument N, and do not return a value to their surrounding context.

Remark 5.4.1. *In terms of literature relating continuations to evaluation contexts, it might be considered more natural to represent the application of a continuation to an argument syntactically as $M[N]$ instead (the insertion of N into the 'hole' of M); we choose not to do so since we extend the standard syntax for $\lambda\mu$.*

It is natural to ask why we represent negation explicitly in the type language, instead of using a type $A\rightarrow\bot$ instead. This is because the ability to distinguish a continuation from a function in terms of the (untyped) syntax is a useful feature when defining the μ-reductions. In particular, a μ-reduction in a function application behaves differently from a μ-reduction in a continuation application, as will become clear in the forthcoming discussions.

It is worth noting that historically, the separation of continuations from functions has also been seen as desirable in more practical settings, for example in Standard ML. As Duba, Harper and McQueen write in [32], "Another way of typing continuations, and the one currently adopted in Standard ML of New Jersey, is to abandon the view that continuations are functions in the ordinary sense ... In practice, it is useful to be able to easily distinguish the invocation of a continuation from the application of a function." In our calculus this is achieved by the continuation applications $[M]N$ as separate entities from the usual function applications $M\ N$. More recently, Kennedy has also argued for the usefulness (particularly in terms of efficiency of optimisations) of a continuation-based intermediate language, in which continuation applications are separated from function applications [52].

5.4.2 Contexts

It will facilitate the discussions of μ-binding to be able to explicitly describe the context in which a μ-bound term occurs: by this we mean a surrounding term with a 'hole', as is described by the following definition:

Definition 5.4.2 (Contexts)**.** *Contexts C are defined using the $\nu\lambda\mu$-syntax, and the special*

symbol $\bullet$ used to denote the (unique) 'hole' in the term:

$$C ::= \bullet \mid C\ M \mid M\ C \mid \lambda x.C \mid [C]M \mid [M]C \mid \nu x.C \mid \mu x.C$$

We write $C\{M\}$ to denote the insertion of the term M into the 'hole' of C, i.e., informally, $C\{M\} = C\langle M/\bullet\rangle$ (however, note that this 'insertion' is allowed to be capturing, unlike our usual substitutions).

For example, we could regard the $\nu\lambda\mu$ term $x\ (\lambda y.((\mu z.M)\ y))$ as $C\{\mu z.M\}$, where C is the context $x\ (\lambda y.(\bullet\ y))$. There are many other ways it could be decomposed as a term inserted into a context.

We note that there is a close relationship between contexts of type $\bot$ and ν-bound terms in our syntax. In fact, for any context C of type $\bot$ and with a 'hole' of type A, the term $\nu x.C\{x\}$ (where x is chosen to be a fresh variable) is of type $\neg A$: the hole in the context is abstracted to form an explicit continuation. Application of this continuation to an argument, i.e. reducing a term of the form $[\nu x.C\{x\}]M$, corresponds exactly with inserting the term M into the context C. Note that an arbitrary term of the form $\nu x.N$ cannot always be related to a context (according to the above definition), since x may occur multiple times.

5.4.3 μ-Reductions

As was discussed in the introduction, there is currently no standard (and obviously complete) set of proof reduction rules for classical natural deduction. This is in contrast to the case of the sequent calculus setting, in which the notion of cut elimination is well-established. In the natural deduction setting, it is the move to classical logic which makes the definition of a canonical set of reduction rules unclear. If one remains in a minimal logic setting, the usual notions of β-style reductions (sometimes extended with η reductions, etc.) will suffice. Since the presence of the μ-binding construct is exactly what makes the $\nu\lambda\mu$-calculus correspond to a classical logic, it seems crucial to choose the reduction rules incorporating this construct (which we refer to as μ-reductions) carefully.

Firstly, we wish to revisit the (μ) reduction rule of the $\lambda\mu$-calculus. Parigot notes in [66] that if one wishes to avoid the special structural substitutions $M\langle[\beta](M'\ N)/[\alpha]M'\rangle$, one can instead employ usual substitution, inserting an abstraction which, after an extra β reduction, has the same effect. This turns out to suggest the following alternative formulation of the rule, adapted for our setting:

Definition 5.4.3 (Alternative formulation of $\lambda\mu$ reduction rule (μ))**.**

$$(\mu x.M)\ N \to \mu y.M\langle \nu z.[y](z\ N)/x\rangle$$

It can be seen that, in the case where the μ-bound variable x occurs only in sub-terms of M of the form $[x]M'$, the effect of this substitution is to replace such subterms with terms of the form $[\nu z.[y](z\ N)]M'$, which in turn reduce (by the rule (ν) given in subsection 5.4.1 above) to terms of the form $[y](M'\ N)$. Therefore the overall effect is similar to that which could be obtained using the structural substitution $M\langle [y](M'\ N)/[x]M'\rangle$. An advantage with this alternative formulation (without structural substitution), is that it works as a reduction rule when x is allowed to be employed in arbitrary positions in M. For example, consider the case $M = [w]x$. Using the reduction rule of Definition 5.4.3 we obtain a perfectly well-defined result, since:

$$([w]x)\langle \nu z.[y](z\ N)/x\rangle = ([w]\nu z.[y](z\ N))$$

On the other hand, the structural substitution $M\langle [y](M'\ N)/[x]M'\rangle$ is not defined to deal with this case. However, Ariola and Herbelin argue [4] that the use of structural substitutions makes for a 'smoother' theory when comparing the calculus with control operators. Furthermore, we find in practice that in the cases where structural substitution could be applied, the extra ν-redexes introduced by taking the approach of Definition 5.4.3 are almost always unwanted, and will be evaluated immediately. We will in fact adopt a middle-ground between these two approaches: essentially an operation which acts as structural substitution when such action is defined, and as normal substitution otherwise. However, for the purposes of the immediate discussions, we will deal just in the style of Definition 5.4.3, and make the necessary modifications later.

We have not, thus far, considered η-like rules for our calculus. However, the rule $(\mu\eta)$, which allows the reduction $\mu x.[x]M \to M$ if $x \notin M$, is included in the original definition of $\lambda\mu$. In fact, this rule turns out to be useful for 'tidying up' after various μ-reductions, and so we are keen to include it. However, since we regard it as an η-like rule, we will postpone the decision of whether or not to include it in our calculus until later in these discussions.

We wish to give two explanations of the general idea we see as underlying the μ-reductions. These focus on the idea that the intended behaviour of μ-binders is to move outwards through the syntax, consuming their outlying context until a point is reached at which they are no longer required.

The first of our explanations focuses on the computational interpretation of a μ-bound

term (we will then move on to discuss the reductions at the level of proofs). For the purposes of these discussions, we would like to imagine we are inventing the μ-binder and μ-reductions from scratch, in order to demonstrate the motivation we see behind the rules. This idea will then generalise to provide the set of μ-reductions we adopt for our calculus.

In order to understand the intended behaviour of a term $\mu x.M$, we find it helpful to examine the form which the body M is allowed to (typeably) take. According to the type system, the typing of M must be of the form: $\Gamma, x : \neg A \vdash M : \bot$. That is, M must be a term of type $\bot$, which itself has a free variable x of type $\neg A$. We can view M as requiring a term of type $\neg A$ (that is, a term which is a continuation with a 'hole' of type A) to replace the variable x with. A 'constructive' way of representing this requirement in the $\nu\lambda\mu$-calculus would be to ν-bind the term M; the term $\nu x.M$ is of type $\neg\neg A$, indicating that it requires an input of type $\neg A$, and, if applied to such a term, will result in type $\bot$. If this approach is taken, the way to then remove the ν-binder on the term would be to use an application of the form $[\nu x.M]N$: i.e., we apply it to a further term N of type $\neg A$, which must be given explicitly in the syntax. However, we consider a 'trick' which is possible: a way of obtaining the desired continuation of type $\neg A$ by more subtle means. Suppose we were to introduce terms of the form $\mu x.M$, and define (arbitrarily, for the purpose of this discussion) that such a term will have type A. Considering the kind of context C in which such a term can be (typeably) placed, it must be a term with a 'hole' of type A itself. Then, starting from the term $C\{\mu x.M\}$, the continuation of type $\neg A$ which M requires could be implicitly defined using whatever the context C is: by ν-abstracting over the 'hole' in the context to form the term $\nu z.C\{z\}$ we can[7] obtain a term of type $\neg A$ suitable to substitute for x. This is in a sense a 'non-constructive' (or at least, indirect) way of specifying the term to insert for x; defined implicitly by the context. We regard this to be the role of the μ-binding in the calculus: to capture its surrounding context, convert it into a ν-abstracted form to explicitly represent it as a continuation, and bind it to a variable. Therefore, we intuitively read terms of the form $\mu x.M$ as "bind the context to x, and evaluate M. Note that this idea is closely related to Bierman's abstract machine for the $\lambda\mu$-calculus, as described in [17].

There are some problems to consider with this point of view. Firstly, in talking about the whole context in which a term is inserted, we lose the notion of local, compatible reductions (such a context could be arbitrarily large, and could itself then be placed in another program). This problem is solved by making a μ-reduction consume only the immediate context, i.e., one level further out in the syntax. Secondly, if this context is not itself of type $\bot$, then it does not represent a continuation in the way we would like.

[7] depending on the type of the whole context - see next paragraph.

Suppose the context is of type B, then we now require a continuation of type $\neg B$ before we can obtain $\bot$. This need for a new continuation can be represented by introducing a new μ-binding appropriately. More generally, one might aim for a local reduction rule to consume the context immediately surrounding a μ-bound term to be defined for every possible syntax construct which may 'sit outside'. Thus, the combination of these rules would allow the μ-binding to progress outwards through the structure of the term, until it reached the outermost level, or (as we will explain) a level at which it is no longer needed. In the special cases of the μ-bound term occurring on the left of a function application, and on the right of a continuation application, we can see this behaviour in the reduction rules of the $\lambda\mu$-calculus. For example, in the rule $(\mu\alpha.M)\ N \rightarrow \mu\beta.M\langle[\beta](M'\ N)/[\alpha]M'\rangle$ (c.f. Definition 5.2.3), the μ-binder initially occurs under an application, whereas after the rule is applied, it is one level further out, in the structure of the term.

With the idea in mind that μ-binders should be propagated outward through the syntax we examine the underlying proofs to see how this might be achieved. The analogous notion here is that instances of the $(\boldsymbol{PC})$ rule should be propagated outward towards the conclusion of a proof. Let us examine, for example, the case of a μ-bound term on the left of a function application (i.e. corresponding to a term of the form $(\mu x.M)\ N$). By examining the type assignment rules, we can see this corresponds to a derivation of the form:

$$\dfrac{\dfrac{\begin{array}{c}M\\ \Gamma, x:\neg(A{\rightarrow}B)\vdash M:\bot\end{array}}{\Gamma\vdash \mu x.M : A{\rightarrow}B}(\boldsymbol{PC}) \qquad \begin{array}{c}N\\ \Gamma\vdash N:A\end{array}}{\Gamma\vdash(\mu x.M)\ N:B}(\rightarrow\mathcal{E})$$

If we wish to move the occurrence of $(\boldsymbol{PC})$ further down, it seems reasonable that we must apply it as the last step; i.e. build a derivation which ends:

$$\dfrac{\Gamma, y:\neg B\vdash\bot}{\Gamma\vdash B}(\boldsymbol{PC})$$

Since we seek a derivation of type $\bot$, it seems that the derivation corresponding to M would be a good candidate. However, this derivation relies on an assumption x of type $\neg(A{\rightarrow}B)$. We have available to us an assumption y of $\neg B$, and also the proof N of A. We notice that $\neg(A{\rightarrow}B)$ is classically equivalent to $A\wedge\neg B$, and so it should be possible to construct a proof of $\neg(A{\rightarrow}B)$ from the assumption $\neg B$ and the proof of A. This proof

can be found and inhabited as follows:

$$\dfrac{\dfrac{\dfrac{}{\Gamma, y:\neg B, z:A\to B \vdash y:\neg B}\,(ax) \qquad \dfrac{\dfrac{}{\Gamma, y:\neg B, z:A\to B \vdash z:A\to B}\,(ax) \qquad \Gamma, y:\neg B, z:A\to B \vdash N:A}{\Gamma, y:\neg B, z:A\to B \vdash z\ N:B}\,(\to\mathcal{E})}{\Gamma, y:\neg B, z:A\to B \vdash [y](z\ N):\bot}\,(\neg\mathcal{E})}{\Gamma, y:\neg B \vdash \nu z.[y](z\ N):\neg(A\to B)}\,(\neg\mathcal{I})$$

If copies of this derivation are now used to replace each occurrence of the assumption x in the derivation corresponding to M (we denote the resulting derivation by M^*), we reach the derivation:

$$\dfrac{\begin{array}{c}\\ M^* \\ \hline \Gamma, y:\neg B \vdash M\langle \nu z.[y](z\ N)/x\rangle : \bot\end{array}}{\Gamma \vdash \mu y.M\langle \nu z.[y](z\ N)/x\rangle : B}\,(\mathit{PC})$$

The argument above derives the reduction rule

$$(\mu\to_1) \quad (\mu x.M)\ N \to \mu y.M\langle \nu z.[y](z\ N)/x\rangle$$

which is exactly that of Definition 5.4.3 (which is itself an alternative formulation of the rule of the original $\lambda\mu$-calculus). So, we have shown a 'first principles' approach for deriving this μ-reduction rule: starting from the objective of permuting (PC) rules further down in a proof, we have obtained the usual reduction rule for the μ-binder.

Applying the same argument to the situation in which the (PC) rule occurs as the *second* premise of an $(\to\mathcal{E})$ rule, i.e., corresponding to starting from a term of the form $N\ (\mu x.M)$, we can derive the following 'symmetrical' reduction rule to the one above:

$$(\mu\to_2) \quad N\ (\mu x.M) \to \mu y.M\langle \nu z.[y](N\ z)/x\rangle$$

This is essentially the rule considered by Parigot in [69] and adopted by various other authors (e.g., [6, 65]).

We can consider other possible inference rules which may occur below an occurrence of (PC) and, taking the same 'first principles' approach, attempt to derive further reduction rules.

For example, consider a term of the form $[N]\mu x.M$. This would represent a derivation of the form:

$$\dfrac{\dfrac{\Gamma \vdash N : \neg A \qquad \dfrac{\Gamma, x : \neg A \vdash M : \bot}{\Gamma \vdash \mu x.M : A}\,(\boldsymbol{PC})}{\Gamma \vdash [N]\mu x.M : \bot}\,(\neg\mathcal{E})}{}$$

As previously, we consider the application of the $(\boldsymbol{PC})$ rule as the last inference step (i.e., we derive the final conclusion $\bot$ using this rule). However, this would result in a derivation ending:

$$\dfrac{\Gamma, y : \neg\bot \vdash \bot}{\Gamma \vdash \bot}\,(\boldsymbol{PC})$$

This seems somewhat counter-intuitive; we obtain $\bot$ from a proof of $\bot$, by contradiction. Furthermore, in a logical sense, the assumption $\neg\bot$ is vacuously true, and so cannot contribute anything meaningful to the subproof above. Instead of taking this approach, we note that M is already a proof of $\bot$, which depends on an assumption x of type $\neg A$. Since N is itself a proof of $\neg A$, N can in fact be inserted for x directly, eliminating the need for the $(\boldsymbol{PC})$ rule altogether. This analysis, along with a similar one of the 'symmetrical' situation (μ-binding on the right of a continuation application) lead to the following two reduction rules (the second of which is standard from $\lambda\mu$, and the first of which is new):

$$\begin{array}{ll} (\mu\neg_1) & [\mu x.M]N \to M\langle \nu z.[z]N/x\rangle \\ (\mu\neg'_2) & [N]\mu x.M \to M\langle N/x\rangle \end{array}$$

There is an asymmetry here, because in the second of the two rules, we have taken a slight 'short-cut': instead of building the term $\nu z.[N]z$ to substitute for x, we have substituted N directly. In terms of the types, this makes no difference. However, for technical reasons, which we will explain in the following chapter (see Section 6.4.2 for details), we choose to abandon this short-cut, and instead adopt the two symmetrical alternatives:

$$\begin{array}{ll} (\mu\neg_1) & [\mu x.M]N \to M\langle \nu z.[z]N/x\rangle \\ (\mu\neg_2) & [N]\mu x.M \to M\langle \nu z.[N]z/x\rangle \end{array}$$

So far, this approach seems to have worked in all cases: by examining the underlying proof structure, we can find a way of rearranging to either move outward or eliminate an occurrence of the $(\boldsymbol{PC})$ rule (correspondingly, of a μ-bound term). What if the μ-bound term occurs under a binder itself? For example, we could form the term $\nu y.\mu x.M$. Is

there a rule for reducing the μ-binder in this situation? Returning to the derivations, we see that we now have one of the form:

$$\dfrac{\dfrac{\Gamma, y : A, x : \neg\bot \vdash M : \bot}{\Gamma, y : A \vdash \mu x.M : \bot}\,(\boldsymbol{PC})}{\Gamma \vdash \nu y.\mu x.M : \neg A}\,(\neg\mathcal{I})$$

As discussed previously, the use of $(\boldsymbol{PC})$ to derive $\bot$ seems to be redundant. We can in fact eliminate this construct entirely: the term $\nu z.z$ represents a canonical proof of type $\neg\bot$, and by replacing x with this term, we can remove the μ-binding; this results in the rule:

$$\nu y.\mu x.M \to \nu y.M\langle \nu z.z/x\rangle$$

The same argument suggests an analogous rule for reducing terms of the form $\mu y.\mu x.M$, i.e.,

$$\mu y.\mu x.M \to \mu y.M\langle \nu z.z/x\rangle$$

The final case to examine is that of μ-binding occurring under a λ-binder (i.e., terms of the form $\lambda y.\mu x.M$. This represents a derivation of the form:

$$\dfrac{\dfrac{\Gamma, y : A, x : \neg B \vdash M : \bot}{\Gamma, y : A \vdash \mu x.M : B}\,(\boldsymbol{PC})}{\Gamma \vdash \lambda y.\mu x.M : A{\to}B}\,(\to\mathcal{I})$$

Our usual technique leads us to seek a derivation ending:

$$\dfrac{\Gamma, z : \neg(A{\to}B) \vdash \bot}{\Gamma \vdash A{\to}B}\,(\boldsymbol{PC})$$

Thus we seek a derivation of $\bot$ from the assumption $\neg(A{\to}B)$. The derivation represented by M would appear to be a possible candidate, since it is of type $\bot$. However, it depends on the two assumptions A and $\neg B$. As commented previously, $\neg(A{\to}B)$ is classically equivalent to $A \wedge \neg B$. Therefore it would seem we can find a suitable rule by deriving proofs for A and $\neg B$ each depending on the assumption of $\neg(A{\to}B)$. Indeed, this is possible, and in the case of $\neg B$ there is no problem. However, in constructing a

proof of A, it turns out that the (PC) rule must be employed (some redundant assumptions have been omitted to save space):

$$\dfrac{\dfrac{}{z:\neg(A\to B), x:\neg A \vdash z:\neg(A\to B)}\,(ax) \qquad \dfrac{\dfrac{\dfrac{}{x:\neg A\vdash x:\neg A}\,(ax) \qquad \dfrac{}{y:A\vdash y:A}\,(ax)}{x:\neg A, y:A\vdash [x]y:\bot}\,(\neg\mathcal{E})}{x:\neg A, y:A\vdash \mu w.[x]y:B}\,(PC)}{x:\neg A\vdash \lambda y.\mu w.[x]y:A\to B}\,(\to\mathcal{I})}{z:\neg(A\to B), x:\neg A\vdash [z]\lambda y.\mu w.[x]y:\bot}\,(\neg\mathcal{E})$$
$$\dfrac{}{z:\neg(A\to B)\vdash \mu x.[z]\lambda y.\mu w.[x]y:A}\,(PC)$$

In other words, by attempting to move the μ-binding outside the λ-binding, we introduce new occurrences of μ-binders under λ-binders into the structure of M. Worse, the occurrences are of the same type, and could also be transformed by such a rule. This leads to non-termination (of typeable terms), and therefore such a rule must be abandoned.

In fact, a weaker rule can be defined, to allow a μ-binding to escape a λ-bound term in the special case of occurring on the left of an application: $(\lambda y.\mu x.M)\ N \to \mu z.[\nu x.M]N$ Although this rule looks somewhat surprising, it is a well-typed reduction, and appears to allow the μ-binding to propagate outward, while leaving the redex essentially in place (the (λ) redex has become a (ν) redex, which can still reduce by the substitution of N for x, as in the original term). However, this rule seems less intuitive than the others we have considered, and does not combine well with the inclusion of the $(\mu\eta)$ rule. A term of the form $\lambda y.\mu x.[x]M$ (with $x \notin M$) can be reduced by the above rule, but if $(\mu\eta)$ was applied first, then no μ reduction rule applies to the reduct $\lambda y.M$.

Instead, inspired by the $\overline{\lambda}\mu\tilde{\mu}$-calculus of Curien and Herbelin [21], we discovered that a better reduction behaviour could be achieved by changing the rule (λ); that is, we modify the original reduction rule of the λ-calculus. We replace the rule (λ) with the following:

$$(\lambda') \quad (\lambda x.M)\ N \to \mu y.[\nu x.[y]M]N$$

This rule looks rather alien to the notion of reduction from the λ-calculus, however we observe that in the presence of the rule $(\mu\eta)$, the original reduction rule can still be simulated:

$$\begin{aligned}(\lambda x.M)\ N &\to \mu y.[\nu x.[y]M]N && (\lambda')\\ &\to \mu y.[y]M\langle N/x\rangle && (\nu)\\ &\to M\langle N/x\rangle && (\mu\eta)\end{aligned}$$

What then, is the advantage of this new version of the β rule? It turns out that, in the presence of the other μ-reduction rules discussed, we can also use it to allow μ-binders to

escape λ-binders:

$$\begin{aligned}(\lambda x.\mu z.M)\ N &\to \mu y.[\nu x.[y]\mu z.M]N && (\lambda')\\ &\to \mu y.[\nu x.M\langle y/z\rangle]N && (\mu\neg_2)\\ &= \mu z.[\nu x.M]N && (\alpha \textit{ conversion})\end{aligned}$$

We therefore adopt this rule, along with $(\mu\eta)$. This completes our analysis of the 'canonical' μ-reduction rules; for each kind of 'context' surrounding a μ-bound term we have attempted to identify a reduction rule which ν-abstracts the context as a (sometimes partial) continuation, and substitutes this new term for the μ-bound variable. This follows our guiding intuition: a μ-bound term essentially binds its surrounding context to a variable, in all situations.

Returning to the question of *structural substitutions*, we can now apply our general view of μ-reductions to the question of the necessity of the ν-abstracted terms which arise out of reduction. In particular, if we consider multiple applications of μ-reduction rules, the case for structural substitution becomes stronger.

Example 5.4.4. *Consider the term $(u\ ((\mu x.[x]M)\ v))$. By applying the μ-reduction rules described above, this can be reduced as follows:*

$$\begin{aligned}(u\ ((\mu x.[x]M)\ v)) &\to (u\ (\mu x.[\nu y.[x](y\ v)]M)) && (\mu\to_1)\\ &\to \mu x.[\nu y.[\nu z.[x](u\ z)](y\ v)]M && (\mu\to_2)\end{aligned}$$

Note that we have reused the binder x at each step, in order to illustrate the 'movement' of the μ-binding; however, these are different binders. The resulting term does not seem very intuitive. In particular, the subterm $\nu y.[\nu z.[x](u\ z)](y\ v)$ is rather cryptic. This is in fact the combination of two 'partial' continuations which have each been constructed by consuming and abstracting one level of the immediate context to the μ-bound term. The redex within this subterm can be reduced by the rule (ν), resulting in the term $\nu y.[x](u\ (y\ v))$. The meaning of this term can be more-readily understood directly; it can be understood to be the context $(u\ (\bullet\ v))$ in which the μ-bound term originally resided, whose 'output' is fed to the continuation variable x (in a continuation application), and whose 'hole' ($\bullet$) has been ν-abstracted, to form a term representing this partially-captured continuation. However, the capture of the context is made even clearer if we reduce both *ν-redexes in the term $\mu x.[\nu y.[\nu z.[x](u\ z)](y\ v)]M$ to reach $\mu x.[x](u\ (M\ v))$. By comparing with the original term $(u\ ((\mu x.[x]M)\ v))$, one can see at this stage that the μ-binding has moved outward, and the context has been consumed into the structure of the term.*

The point of this example is to illustrate that it is only after the reduction of the newly-

created (ν)-redexes that the resulting terms can be seen to neatly correspond to our intuitive semantics for the μ-reductions. Therefore, it seems that structural substitution would be a more natural choice than the creation of new ν-bindings and redexes. This is an informal observation in support of the conclusion of [4]; that structural substitution makes for a neater reduction behaviour. However, since one of our stated aims was to retain a full correspondence with the original logic, we do not wish to restrict occurrences of μ-bound variables, in the way the $\lambda\mu$-calculus does. Instead, we define a notion of substitution 'in between' the two approaches; since our objection to structural substitution is that it cannot be applied in all cases, we apply it when it can, and introduce explicit ν-bindings when it cannot. This idea is formalised as a new meta operation, which we call *semi-structural substitution*.

Definition 5.4.5 (Semi-Structural Substitution)**.** *We define the operation $M\langle\hat{y}(N)/x\rangle$ recursively over the structure of M, by the following rules:*

$$\begin{aligned}
x\langle\hat{z}(N)/x\rangle &= \nu z.N \\
y\langle\hat{z}(N)/x\rangle &= y && \text{if } y \neq x \\
([x]M)\langle\hat{z}(N)/x\rangle &= N\langle M\langle\hat{z}(N)/x\rangle/z\rangle \\
([M_1]M_2)\langle\hat{z}(N)/x\rangle &= [M_1\langle\hat{z}(N)/x\rangle]M_2\langle\hat{z}(N)/x\rangle && \text{if } M_1 \neq x \\
(M_1\ M_2)\langle\hat{z}(N)/x\rangle &= M_1\langle\hat{z}(N)/x\rangle\ M_2\langle\hat{z}(N)/x\rangle \\
(\nu y.M)\langle\hat{z}(N)/x\rangle &= \nu y.M\langle\hat{z}(N)/x\rangle \\
(\lambda y.M)\langle\hat{z}(N)/x\rangle &= \lambda y.M\langle\hat{z}(N)/x\rangle \\
(\mu y.M)\langle\hat{z}(N)/x\rangle &= \mu y.M\langle\hat{z}(N)/x\rangle
\end{aligned}$$

We can then formulate the reduction rules discussed above using this operation; wherever we would have substituted a ν-bound term, i.e., applied a substitution of the form $M\langle\nu z.N/x\rangle$, we instead apply a semi-structural substitution of the form $M\langle\hat{z}(N)/x\rangle$.

The complete set of reduction rules for the calculus are as follows.

Definition 5.4.6 (Reduction rules for the $\nu\lambda\mu$-calculus)**.** *In the following rules, all variable names occurring on the right but not the left of a reduction rules are assumed to be*

fresh (these correspond to new binders, and must not introduce clashes/capturing).

$$
\begin{array}{lll}
(\lambda') & (\lambda x.M)\ N \to \mu y.[\nu x.[y]M]N & \\
(\nu) & [\nu x.M]N \to M\langle N/x\rangle & \\
(\mu\to_1) & (\mu x.M)\ N \to \mu y.M\langle\hat{z}([y](z\ N))/x\rangle & \\
(\mu\to_2) & N\ (\mu x.M) \to \mu y.M\langle\hat{z}([y](N\ z))/x\rangle & \\
(\mu\neg_1) & [\mu x.M]N \to M\langle\hat{z}([z]N)/x\rangle & \\
(\mu\neg_2) & [N]\mu x.M \to M\langle\hat{z}([N]z)/x\rangle & \\
(\mu\nu) & \nu y.\mu x.M \to \nu y.M\langle\hat{z}(z)/x\rangle & \\
(\mu\mu) & \mu y.\mu x.M \to \mu y.M\langle\hat{z}(z)/x\rangle & \\
(\mu\eta) & \mu x.[x]M \to M & \textit{if } x \notin \textit{fv} M
\end{array}
$$

As usual, we define our reduction relation $\to$ to be the reflexive, transitive, compatible closure of the above rules. Although we have now formally replaced the (λ) rule previously discussed, we will treat it as an admissible rule, as is justified by the above discussion (and so still allow ourselves to use it for reduction), in order to shorten the examples we require later.

The reductions defined above are sound with respect to the type system (Definition 5.3.2):

Proposition 5.4.7 (Substitution lemmas and subject reduction)**.**

1. *If* $x \notin \Gamma$ *and* $\Gamma \vdash M : A$ *and* $\Gamma, x : A \vdash N : B$*, then* $\Gamma \vdash N\langle M/x\rangle : B$.
2. *If* $x, z \notin \Gamma$ *and* $\Gamma \vdash M : \bot$ *and* $\Gamma, x : \neg A \vdash N : B$*, then* $\Gamma \vdash N\langle\hat{z}(M)/x\rangle : B$.
3. *If* $\Gamma \vdash M : A$ *and* $M \to N$ *then* $\Gamma \vdash N : A$.

Proof. 1. By straightforward induction on the structure of the term N.

2. By straightforward induction on the structure of the term N, using part 1.

3. By straightforward induction on the length of the reduction sequence, and the structure of the term M, using parts 1 and 2.

□

5.5 Examples

5.5.1 Example: Representing Pairing

As a simple example of the syntax and reductions of the $\nu\lambda\mu$-calculus, we show how to encode *pairing*, by providing a representation for pairs of terms $\langle M,N\rangle$, and their usual two projection operators (*fst* and *snd*). These constructs correspond logically to the rules for the conjunction ($\wedge$) connective, which is not treated as a primitive connective in our logical setting. Instead, we make use of the classical equivalence of the formulae $A\wedge B$ and $\neg(A\rightarrow\neg B)$.

Our representations are as follows (in which all variable names mentioned are assumed to be fresh, i.e. not occurring in M,N):

$$\begin{aligned}\langle M,N\rangle &= \nu w.[w\ M]N\\ \mathit{fst}(M) &= \mu x.[M]\lambda y.\mu z.[x]y\\ \mathit{snd}(M) &= \mu x.[M]\lambda y.x\end{aligned}$$

These come from the following derivations:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{w:A\rightarrow\neg B\vdash w:A\rightarrow\neg B}\,(ax) \qquad w:A\rightarrow\neg B\vdash M:A}{w:A\rightarrow\neg B\vdash w\ M:\neg B}\,(\rightarrow\mathcal{E}) \qquad w:A\rightarrow\neg B\vdash N:B}{w:A\rightarrow\neg B\vdash[w\ M]N:\bot}\,(\neg\mathcal{E})}{\vdash\nu w.[w\ M]N:\neg(A\rightarrow\neg B)}\,(\neg\mathcal{I})$$

$$\dfrac{x:\neg A\vdash M:\neg(A\rightarrow\neg B) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{x:\neg A,y:A,z:\neg\neg B\vdash x:\neg A}\,(ax) \qquad \dfrac{}{x:\neg A,y:A,z:\neg\neg B\vdash y:A}\,(ax)}{x:\neg A,y:A,z:\neg\neg B\vdash[x]y:\bot}\,(\neg\mathcal{E})}{x:\neg A,y:A\vdash\mu z.[x]y:\neg B}\,(PC)}{x:\neg A\vdash\lambda y.\mu z.[x]y:A\rightarrow\neg B}\,(\rightarrow\mathcal{I})}{x:\neg A\vdash[M]\lambda y.\mu z.[x]y:\bot}\,(\neg\mathcal{E})}{\vdash\mu x.[M]\lambda y.\mu z.[x]y:A}\,(PC)$$

$$\dfrac{\dfrac{x:\neg B\vdash M:\neg(A\rightarrow\neg B) \qquad \dfrac{\dfrac{}{x:\neg B\vdash x:\neg B}\,(ax)}{x:\neg B\vdash\lambda y.x:A\rightarrow\neg B}\,(\rightarrow\mathcal{I})}{x:\neg B\vdash[M]\lambda y.x:\bot}\,(\neg\mathcal{E})}{\vdash\mu x.[M]\lambda y.x:B}\,(PC)$$

Our term representations are the inhabitants of the appropriate canonical natural deduction proofs. For example, the simplest proof of the formula $\neg(\neg A\rightarrow B)$ from assumptions A and B is inhabited by the $\nu\lambda\mu$-term $\nu w.[w\ M]N$, as shown above. To show that we

can simulate the expected behaviour, we must verify that *fst*$(\langle M,N\rangle) \to M$ and then *snd*$(\langle M,N\rangle) \to N$. This is demonstrated as follows:

$$\begin{aligned}
\textit{fst}(\langle M,N\rangle) &= \mu x.[\nu w.[w\ M]N]\lambda y.\mu z.[x]y \\
&\to \mu x.[(\lambda y.\mu z.[x]y)\ M]N && (\nu) \\
&\to \mu x.[\mu z.[x]M]N && (\lambda) \\
&\to \mu x.[x]M && (\mu\neg_1) \\
&\to M && (\mu\eta)
\end{aligned}$$

$$\begin{aligned}
\textit{snd}(\langle M,N\rangle) &= \mu x.[\nu w.[w\ M]N]\lambda y.x \\
&\to \mu x.[(\lambda y.x)\ ]MN && (\nu) \\
&\to \mu x.[x]N && (\lambda) \\
&\to N && (\mu\eta)
\end{aligned}$$

Remark 5.5.1. *It is interesting to note that these reductions need not be deterministic, highlighting the non-confluent nature of the calculus. In fact, it is possible that, because of some effect stemming from one of the terms M or N, the reduction behaviour can potentially be quite different from that shown above. For example, if N were of the form $\mu v.N'$, with $v \notin N'$ (i.e., N can be read as $\mathcal{A}(N')$, a term whose behaviour is to 'abort' the surrounding context, as will be explained in Chapter 6), then the term representing fst$(\langle M,N\rangle)$ can also reduce to N (essentially, M is discarded by evaluating the 'abort' first):*

$$\begin{aligned}
\textit{fst}(\langle M,\mu v.N'\rangle) &= \mu x.[\nu w.[w\ M]\mu v.N']\lambda y.\mu z.[x]y \\
&\to \mu x.[\nu w.N']\lambda y.\mu z.[x]y && (\mu\neg_2) \\
&\to \mu x.N' && (\nu) \\
&= N
\end{aligned}$$

Note that the last line is by α-conversion, since x does not occur in N. This non-confluence is a natural consequence of an unrestricted canonical notion of reduction for classical logic (and also for some notions of control, as will be argued in the next chapter).

5.5.2 Encoding the λ-calculus

A natural example to examine is the encoding of the λ-calculus in our new calculus. We will show not only that we can encode the calculus, preserving reductions and typings, but that although $\nu\lambda\mu$ is non-confluent, the image of the λ-calculus in $\nu\lambda\mu$ is confluent. The

encoding of a λ-calculus term M is written $\overline{M}$ and defined recursively by the 'obvious' injection:

$$\begin{aligned} \overline{x} &= x \\ \overline{\lambda x.M} &= \lambda x.\overline{M} \\ \overline{M\ N} &= \overline{M}\ \overline{N} \end{aligned}$$

For convenience, we choose not to distinguish syntactically between λ-calculus terms and their counterparts in the $\nu\lambda\mu$ calculus. For example, if $\lambda x.t$ is a term of the λ-calculus, we regard it also as a term of $\nu\lambda\mu$, for purposes of our discussions.

In the following discussions, we write $t_1 \rightarrow_\beta t_2$ for the reflexive, transitive, contextual closure of the usual one-step reduction relation of the λ-calculus (β-reductions only), and $=_\beta$ for the reflexive, transitive, symmetric, compatible closure of this relation. We will make use of the following standard result (slightly restated):

Lemma 5.5.2 (Confluence of λ-calculus)**.** *For any two λ-calculus terms t_1,t_2, if $t_1 =_\beta t_2$ then there exists a λ-calculus term t such that $t_1 \rightarrow_\beta t$ and $t_2 \rightarrow_\beta t$.*

As was essentially described above, although the (β) rule of the λ-calculus is not present in our reduction rules for $\nu\lambda\mu$, it can be simulated, as the following result makes clear.

Proposition 5.5.3 (Simulation of the λ-calculus)**.** *For any λ-calculus terms t_1, t_2, if $t_1 \rightarrow_\beta t_2$ then $t_1 \rightarrow_{\nu\lambda\mu} t_2$.*

Proof. Since both reduction relations are compatible, we need only check the (β) rule directly:

$$\begin{aligned} (\lambda x.M)\ N &\rightarrow \mu y.[\nu x.[y]M]N && (\lambda) \\ &\rightarrow \mu y.[y]M\langle N/x\rangle && (\nu) \\ &\rightarrow M\langle N/x\rangle && (\mu\eta) \end{aligned}$$

□

It is interesting to examine whether or not the image of the λ-calculus defines a confluent subset of the $\nu\lambda\mu$ calculus. In other words, is it possible to interpret a λ-calculus term in our setting, and then run it in ways which are essentially different from those of the original? Intuitively it seems this should not be possible: since we start in the realm of a minimal natural deduction proof, and have a subject reduction property, we expect to stay in this realm. To take the same idea from a computational perspective, if we start from a term without control behaviour, we would not expect it to become a term with control behaviour during reduction.

We wish to consider the (smallest) subset of the $\nu\lambda\mu$ syntax including the image of the λ-calculus and closed under reduction. We will call this sub-syntax of the $\nu\lambda\mu$-syntax Λ. We will use m,n to range over the terms in this sub-syntax Λ. After some experimentation, various properties can be observed, which show this subset is a true restriction. One key fact is that μ-bound variables always occur linearly (exactly one occurrence is bound). This means that effects such as duplicating and discarding contexts/continuations are no longer available. Because of this linear nature of the μ-binder, the μ reductions actually become rather weak: our intuition was that they essentially only move parts of the context around in this restricted setting, before the 'real' evaluations are eventually performed[8]. In order to make this idea more concrete, we will give encodings from Λ back to the pure syntax of the λ-calculus (in which only the 'real' reductions take place). Firstly, we make the following observations about the restricted syntax.

Proposition 5.5.4 (Λ is a true restriction of $\nu\lambda\mu$)**.** *The following properties hold for the sub-syntax Λ:*

1. *μ-bound variables x always occur linearly in terms (there is exactly one occurrence bound).*
2. *μ-bound terms are always of the form $\mu x.[m]n$, and ν-bound terms are always of the form $\nu x.[m]n$.*
3. *Conversely, terms of the form $[m]n$ can only occur as the body of a μ or ν-binding.*
4. *Terms of the form $[m]n$ can only occur when m is either a μ-bound variable x or m is a term of the form $\nu y.[m_1]m_2$ (in which the form of m_1 is subject to the same restrictions).*
5. *Conversely, μ-bound variables x, and terms of the form $\nu x.[m_1]m_2$ can only occur on left of terms of the form $[m]n$.*
6. *In terms of the form $\nu x.[m_1]m_2$, x never occurs in m_1.*
7. *In terms of the form $\mu x.[m_1]m_2$, the (unique) occurrence of x is always in m_1.*

Proof. All of the conditions are trivially satisfied by the encoding of a λ-calculus term itself. We illustrate that these properties are preserved by reduction, with the example of the $(\mu\rightarrow_1)$ rule.

[8]Berdine et. al. [16] present work on the restriction of continuations to linear occurrences, which they argue are sufficient for many of the "behind the scenes" applications of continuations in, for example, implementations of other language features.

Suppose then, that the properties all hold for a term of the form $(\mu x.m)\ n$. Since property 2 holds for this term, m must be of the form $[m_1]m_2$ for some terms m_1,m_2. The reduct of the rule is the term $\mu y.[m_1]m_2\langle \nu z.[y](z\ n)/x\rangle$. Property 1 can be seen to be preserved; since x occurred linearly in the original term then the substitution is made in exactly one place, and exactly one occurrence of y results. Properties 2 and 3 can be seen from the form of the reduct. Properties 4 and 5 can be seen to be preserved since the substitution replaces a μ-bound variable with a term of the other permissible form. Property 6 still holds because of the form of the term being substituted in place of x. Property 7 is preserved since the substitution inserts the unique occurrence of y (within a term) in place of the unique occurrence of x. □

In fact, the last of these restrictions can be deduced from the others, in the following way. Firstly, given the restriction on occurrences of ν-binders and terms of the form $\nu m.n$, it is apparent that the outermost syntax construct of a term must either be a construct from the λ-calculus, or a μ-binder. Assume that we eventually reach a μ-bound term by traversing down the structure of the term, and that it is of the form $\mu x.[m]n$. By the conditions above, m is either a μ-bound variable, in which case it must be x, or else m is of the form $\nu y.[m_1]m_2$. However, this gives a new continuation application to which property 4 applies. In order for this analysis to terminate, the μ-bound variable x must eventually occur on the left of one of these applications. Since it occurs linearly, it must never occur on the right of one of these applications (property 7). However, since no further opportunity for μ-binders to occur has been reached, the terms occurring on the left of $[m]n$ terms (in *context* position in these continuation applications) are of a very specific form, which depends on the name of unique enclosing μ-bound variable. We characterise these particular terms by defining, for each variable x the set of 'contexts depending on x', ranged over by $c_{(x)}$. This allows us to specify our restricted syntax more precisely:

Definition 5.5.5 (Characterisation of terms of Λ)**.** *The terms of the subset Λ are defined by the grammar below. We use $c_{(x)}$ to range over 'contexts defined over x':*

$$
\begin{array}{rcll}
m, n & ::= & y & \\
 & | & \lambda y.m & \\
 & | & m\ n & \\
 & | & \mu y.[c_{(y)}]m & (y \notin m) \\
 & & & \\
c_{(x)} & ::= & x & \\
 & | & \nu y.[c_{(x)}]m & (y \notin c_{(x)})
\end{array}
$$

So far, this doesn't give a very clear picture of what this restricted syntax might mean

in terms of the original λ-calculus. However, below we will show that encodings from this syntax back to the pure λ-calculus are possible, with useful properties. Our method was inspired by the work of Bierman [17], who gave a semantics for the $\lambda\mu$-calculus in terms of evaluation contexts. The extension of this work to the full $\nu\lambda\mu$-calculus is not obviously possible, since it relies heavily on the confluent nature of $\lambda\mu$. However, by following the belief that the sub-syntax we are now dealing with is now confluent, we borrow from his ideas.

The basic idea behind Bierman's abstract machine is that when a $\lambda\mu$-term of the form $\mu\alpha.M$ is to be evaluated in a context $E\{.\}$, the context $E\{.\}$ is 'stored' and bound to α, and then the term M evaluated. If subterms of the form $[\alpha]N$ are encountered, the context $E\{.\}$ is restored, i.e., one evaluates $E\{N\}$ at this point. Following this idea (and as discussed previously), one can read terms of the form $[m]n$ as 'evaluate n in the context denoted by m'. In our setting this requires a generalisation of Bierman's idea, since it is possible for terms of the form $[m]n$ to occur in which m is not a μ-bound variable. However, as identified in the restrictions above, the only other possibility which arises is that m is of the form $\nu y.[c_{(x)}]m_2$, i.e., we wish to give an interpretation for the term $[\nu y.[c_{(x)}]m_2]n$. Since y occurs only in m_2, and bearing in mind that the (ν) redex provides the facility to substitute copies of n for the ys in m_2, we choose to read this term as 'evaluate *let* $x = n$ *in* m_2 in the context $c_{(x)}$'. This gives an idea for how to retrieve a term closer to one of the λ-calculus instead (essentially by an expansion of the 'let' into a β-redex); the term $[c_{(x)}]((\lambda y.m_2)\ n)$ is a term which we would intuitively read in a similar way. By recursively replacing all terms of the form $[\nu y.[c_{(x)}]m_2]n$ with the corresponding terms $[c_{(x)}]((\lambda y.m_2)\ n)$, we eventually obtain terms in which the only contexts $c_{(x)}$ are x itself. This means that the only μ-bound terms are of the form $\mu x.[x]m$, in which m is a term of the λ-calculus with $x \notin m$. By finally replacing terms of this form with simply m itself (c.f. the $(\mu\eta)$ rule), we reach terms of the pure λ-calculus syntax. This idea is formalised by the following definition:

Definition 5.5.6 (Λ back to the λ-calculus (with expansions))**.** *We use t, t_1, t_2 to range over terms in the $\nu\lambda\mu$ syntax which are also terms of the λ-calculus. We define two mutually recursive mappings: $(m)^\circ$ which maps terms of Λ to λ-calculus terms, and $(m)^\bullet$ which maps terms of the form $[c_{(x)}]t$ (where t is a term of the λ-calculus) into terms of the form $[x]t_2$ where t_2 is also a (possibly different) term of the λ-calculus.*

The definitions are as follows:

$$
\begin{aligned}
(x)^\circ &= x \\
(\lambda x.m)^\circ &= \lambda x.(m)^\circ \\
(m\ n)^\circ &= (m)^\circ\ (n)^\circ \\
(\mu x.[c_{(x)}]m)^\circ &= t \\
&\textit{where} \\
[x]t &= ([c_{(x)}](m)^\circ)^\bullet
\end{aligned}
$$

$$
\begin{aligned}
([x]t)^\bullet &= [x]t \\
([\nu y.[c_{(x)}]m]t)^\bullet &= ([c_{(x)}]((\lambda y.(m)^\circ)\ t))^\bullet
\end{aligned}
$$

The following proposition essentially states that the mappings above are well-defined:

Proposition 5.5.7. *1. The evaluation of $(m)^\circ$ and $([c_{(x)}]t)^\bullet$ (by the rules defined above) always terminates.*

2. $([c_{(x)}]t_1)^\bullet$ always evaluates to a term of the form $[x]t_2$ (where t_1,t_2 are terms of the λ-calculus).

Proof. 1. By mutual induction on the structure of the terms m featuring in $(m)^\circ$ and the contexts $c_{(x)}$ featuring in $([c_{(x)}]t)^\bullet$.

2. By induction on the definition of $([c_{(x)}]t)^\bullet$ (using part 1 to ensure well-foundedness).

□

What has been done here? Essentially, we have restored all portions of context which have been shifted by μ reductions back to their original positions, and then applied expansions to restore λ-calculus redexes in the position of (ν) ones. Once the contexts are all back in place, all the μ-bindings are of the trivial kind (to which the $(\mu\eta)$ rule applies) and can be eliminated. The idea is that the λ-calculus term we end up with represents 'essentially' where we have got to so far in the computation, without the complications of rearranged contexts, etc. Many of the μ-reductions turn out to be transparent under the mapping above (i.e., both redex and redex are mapped to the same λ term), although some result in a β-expansion. The (ν) reductions on the other hand make actual substitutions, and these correspond to (β) reductions in the λ-calculus. Thus, we have a correspondence between the two reduction relations, but expansions are sometimes required on one side or another.

We can emphasise the correspondence between the reduction relations by the following result:

Lemma 5.5.8. *For any terms m,n of the sub-syntax Λ, if $m \to_{\nu\lambda\mu} n$ then $(m)^\circ =_\beta (n)^\circ$.*

In order to give a stronger correspondence between reductions in our restricted $\nu\lambda\mu$, and those of the λ-calculus, we define a second pair of mappings. The idea now is, rather than to create the new λ-calculus redexes in the case $([\nu y.[c_{(x)}]m]t)^\bullet$, we evaluate them. That is, instead of $(\lambda y.(m)^\circ)\ t$ we use $(m)^\circ\langle t/y\rangle$. The advantage here is that the resulting term can actually be reached by $\nu\lambda\mu$ reductions.

The definitions of these modified mappings are otherwise similar to those given above, but we include them for reference:

Definition 5.5.9 (Λ back to the λ-calculus (with reductions))**.**

$$\begin{aligned}
(x)^{\circ\circ} &= x \\
(\lambda x.m)^{\circ\circ} &= \lambda x.(m)^{\circ\circ} \\
(m\ n)^{\circ\circ} &= (m)^{\circ\circ}\ (n)^{\circ\circ} \\
(\mu x.[c_{(x)}]m)^{\circ\circ} &= t \\
&\textit{where} \\
[x]t &= ([c_{(x)}](m)^{\circ\circ})^{\bullet\bullet}
\end{aligned}$$

$$\begin{aligned}
([x]t)^{\bullet\bullet} &= [x]t \\
([\nu y.[c_{(x)}]m]t)^{\bullet\bullet} &= ([c_{(x)}](m)^{\circ\circ}\langle t/y\rangle)^{\bullet\bullet}
\end{aligned}$$

We can now set up the remaining framework required to prove confluence of the subset Λ. Firstly, we observe that since the only difference between the mappings $(m)^\circ$ and $(m)^{\circ\circ}$ is that extra β-reductions have taken place in the result of the latter, we know in particular that their results are always $=_\beta$ to one another:

Lemma 5.5.10. *For any term m of Λ, we have $(m)^\circ =_\beta (m)^{\circ\circ}$.*

Proof. By straightforward induction on the definition of $(m)^{\circ\circ}$ (which is well-founded, by an argument similar to that of Proposition 5.5.7(1)). □

The advantage of the second mapping is that it provides a closer correspondence between the two reduction relations. On the one hand, the changes that the mapping itself makes to an input term can be simulated by the reductions of $\nu\lambda\mu$. On the other hand, if one interprets a term via the mapping and then performs λ-calculus reductions on the result, these can also be simulated in the $\nu\lambda\mu$ calculus.

Lemma 5.5.11. *For any term m of Λ, we have $m \to (m)^{\circ\circ}$.*

Proof. By straightforward induction on the definition of $(m)^{\circ\circ}$. □

Note that an analogous result would not have been possible for the first mappings defined, since expansions take place to restore the λ-calculus redexes. In fact, one could have done without these mappings at all, but the proof of an analogous result to Lemma 5.5.8 is then harder, since there is more work to do in one result.

We are now in a position to prove our confluence result:

Theorem 5.5.12 (The image of the λ-calculus is confluent in $\nu\lambda\mu$)**.** *Let m,m_1,m_2 be any terms in the sub-syntax Λ such that $m \rightarrow m_1$ and $m \rightarrow m_2$. Then there exists a term $n \in \Lambda$ such that $m_1 \rightarrow n$ and $m_2 \rightarrow n$.*

Proof. By applying Lemma 5.5.8 twice, we have that $(m_1)^{\circ} =_\beta (m)^{\circ}$ and that $(m)^{\circ} =_\beta (m_2)^{\circ}$. By Lemma 5.5.10 twice, we obtain $(m_1)^{\circ\circ} =_\beta (m_2)^{\circ\circ}$. By Lemma 5.5.2, there exists a term t of the λ-calculus such that $(m_1)^{\circ\circ} \rightarrow_\beta t$ and $(m_2)^{\circ\circ} \rightarrow_\beta t$. By applying Proposition 5.5.3 in both cases, we have $(m_1)^{\circ\circ} \rightarrow_{\nu\lambda\mu} t$ and $(m_2)^{\circ\circ} \rightarrow_{\nu\lambda\mu} t$. Finally, by Lemma 5.5.11, we have $m_1 \rightarrow_{\nu\lambda\mu} (m_1)^{\circ\circ} \rightarrow_{\nu\lambda\mu} t$ and $m_2 \rightarrow_{\nu\lambda\mu} (m_2)^{\circ\circ} \rightarrow_{\nu\lambda\mu} t$. We conclude, taking n to be t. □

Note that we actually show not only that m_1 and m_2 are joinable, but that they can always be made to join on a term of the λ-calculus. In particular, this implies that the normal forms in Λ are all terms of the λ-calculus; all other syntax constructs only occur in intermediate reductions.

5.5.3 Simulation of $\lambda\mu$

We show that the $\lambda\mu$-calculus can be encoded into the $\nu\lambda\mu$ calculus by a simple injection. Since this calculus deals with two separate classes of variables, for ease of definition we extend the (single) class of variables in $\nu\lambda\mu$ to include Greek characters also.

Definition 5.5.13 (Encoding $\lambda\mu$)**.** *We encode $\lambda\mu$ into $\nu\lambda\mu$ in the following way:*

$$
\begin{aligned}
\llbracket x \rrbracket_{\lambda\mu} &= x \\
\llbracket \lambda x.M \rrbracket_{\lambda\mu} &= \lambda x.\llbracket M \rrbracket_{\lambda\mu} \\
\llbracket M\ N \rrbracket_{\lambda\mu} &= \llbracket M \rrbracket_{\lambda\mu}\ \llbracket N \rrbracket_{\lambda\mu} \\
\llbracket [\alpha]M \rrbracket_{\lambda\mu} &= [\alpha]\llbracket M \rrbracket_{\lambda\mu} \\
\llbracket \mu\alpha.M \rrbracket_{\lambda\mu} &= \mu\alpha.\llbracket M \rrbracket_{\lambda\mu}
\end{aligned}
$$

As in the previous sections, we need to define, for any right-context Δ (of a $\lambda\mu$ typing), $\neg\Delta = \{\alpha : \neg A \mid \alpha : A \in \Delta\}$. We can then state the following results for our encoding.

Proposition 5.5.14 (Simulation of $\lambda\mu$)**.** *1. The mapping $[\![.]\!]_{\lambda\mu}$ is an injection.*

2. *For any $\lambda\mu$ terms M,N, if $M \to_{\lambda\mu} N$ then $[\![M]\!]_{\lambda\mu} \to [\![N]\!]_{\lambda\mu}$.*

3. *For any $\lambda\mu$ term M, if $\Gamma \vdash_{\lambda\mu} M : A \mid \Delta$ then $\Gamma, \neg\Delta \vdash [\![M]\!]_{\lambda\mu} : A$.*

Proof. 1. By inspection of Definition 5.5.13, we can see that the encoding is compositional, and maps terms with distinct top-level syntax constructs onto terms with distinct top-level syntax constructs. Therefore, the result follows by a straightforward induction on the structure of terms.

2. To show that reduction is preserved, we could present a full inductive argument. However, by inspection of Definitions 5.2.3 and 5.4.6, it is clear that the $\lambda\mu$ reduction rules are closely related to a subset of the $\nu\lambda\mu$ reduction rules. There are three significant discrepancies:

   (a) The $\lambda\mu$-calculus includes the normal (β)-rule, whereas we have our modified (λ'). However, the argument of Proposition 5.5.3 shows that the former can be simulated by the latter.

   (b) The (μ) rule of $\lambda\mu$ corresponds to the $(\mu{\to_1})$ rule of $\nu\lambda\mu$, but employs structural substitution, whereas we employ semi-structural substitution. However, since α may only occur in positions $[\alpha]N$ in $\lambda\mu$, in these cases the two operations coincide (see Definition 5.4.5).

   (c) The (μr) rule of $\lambda\mu$ corresponds to the $(\mu{\neg_2})$ rule of $\nu\lambda\mu$, but the substitutions employed differ; in $\lambda\mu$ the reduct of $[\beta]\mu\alpha.M$ is $M\langle\beta/\alpha\rangle$, whereas in $\nu\lambda\mu$ it would be $M\langle\hat{z}([\beta]z)/\alpha\rangle$. However, again we note that occurrences of α in M are syntactically restricted to be in subterms of the form $[\alpha]N$. By Definition 5.4.5, we have $([\alpha]N)\langle\hat{z}([\beta]z)/\alpha\rangle = ([\beta]z)\langle N/z\rangle = ([\beta]N)$; i.e. the same reduct is reached as in $\lambda\mu$. In more general terms, since semi-structural substitution is defined to coincide with structural substitution in the restricted cases which occur in $\lambda\mu$, this difference in the presentation of the rules does not affect our simulation result.

3. By straightforward induction on the structure of the derivation $\Gamma \vdash_{\lambda\mu} M : A \mid \Delta$.

□

As an example of the differences between these two calculi, we consider terms in each which might be seen to have type $\neg\neg A \to A$. Ariola and Herbelin [3] discuss issues regarding the inhabitation of this type in $\lambda\mu$: "In Parigot's style... [this type]... is represented with the term $\lambda y.\mu\alpha.[\gamma](y\ (\lambda x.\mu\delta.[\alpha]x))$". This term is significantly more complex than might be expected of a canonical term inhabiting this type, which reflects the need to work around the restrictions on the form of terms involving μ-variables. This is improved upon by allowing the body of a μ-abstraction to be any term of type $\bot$ [73, 17], permitting a term such as $\lambda y.\mu\alpha.(y\ (\lambda x.[\alpha]x))$ instead. However, the canonical natural deduction proof of $\neg\neg A \to A$ yields the $\nu\lambda\mu$-term $\lambda y.\mu x.[y]x$. This cannot be a $\lambda\mu$ term, because the μ-bound variable is used as a standard term variable. Therefore we have a simpler representation than in other comparable calculi. This term has behaviour similar (although not identical, as we shall see) to the Felleisen's $\mathcal{C}$ operator: when applied to an argument (bound to y), it captures the outlying context (binding it to x), and then passes x to y in a continuation application. The representation of control operators will be discussed further in the next chapter.

5.5.4 The $\overline{\lambda}\mu\tilde{\mu}$-Calculus

In this section, we show that we can encode the $\overline{\lambda}\mu\tilde{\mu}$-calculus into $\nu\lambda\mu$. This is perhaps more surprising than our previous simulation results, since $\overline{\lambda}\mu\tilde{\mu}$ is based on a sequent calculus presentation of classical logic. The $\overline{\lambda}\mu\tilde{\mu}$-calculus [21] has a Curry-Howard correspondence with a modified version of Gentzen's LK (sequent calculus for classical logic). The modification generalises the usual sequents $\Gamma \vdash \Delta$ to allow an optional distinguished statement on the left or right of the turnstile (but never both at once). If such a formula is present, it is written between the turnstile and a *stoup* ($|$). These three possible kinds of sequents give rise to three different classes of syntax: terms (which correspond to sequents with a distinguished formula on the right), contexts (with a distinguished formula on the left) and commands (the original kind of sequent, with no distinguished formula).

Definition 5.5.15 ($\overline{\lambda}\mu\tilde{\mu}$ syntax)**.** *The syntax of the terms (ranged over by v), contexts (ranged over by e) and commands (ranged over by c) of the $\overline{\lambda}\mu\tilde{\mu}$-calculus is specified by the following mutually-recursive definitions, in which x, y range over term variables, and α, β over context variables:*

$$\begin{aligned} c &:= \langle v \mid e \rangle \\ v &:= x \mid \lambda x.v \mid \mu\alpha.c \\ e &:= \alpha \mid v \cdot e \mid \tilde{\mu}x.c \end{aligned}$$

The commands $\langle v \mid e \rangle$ correspond to cuts in the logical sense, and reduction induces a partial cut-elimination. However, not all cuts are redexes in this calculus: the command

$\langle x \mid \alpha \rangle$ is in normal form, for example. Commands pair a term with a context, and evaluation of such a command intuitively defines what it means to evaluate the term in the context. However, this intuition can be slightly misleading; it is not always the term which is inserted into the context and sometimes the dual behaviour can be observed. The full reduction rules are defined as follows:

Definition 5.5.16 ($\overline{\lambda}\mu\tilde{\mu}$ reductions)**.** *The reduction relation of the $\overline{\lambda}\mu\tilde{\mu}$-calculus (which we write as $\rightarrow_{\overline{\lambda}\mu\tilde{\mu}}$ is defined to be the reflexive, transitive, compatible closure of the following rules, and is defined in terms of the usual notion of implicit substitution.*

$$\begin{array}{ll} (\rightarrow') & \langle \lambda x.v_1 \mid v_2 \cdot e \rangle \rightarrow_{\overline{\lambda}\mu\tilde{\mu}} \langle v_2 \mid \tilde{\mu}x.\langle v_1 \mid e \rangle \rangle \\ (\mu) & \langle \mu\beta.c \mid e \rangle \rightarrow_{\overline{\lambda}\mu\tilde{\mu}} c\langle e/\beta \rangle \\ (\tilde{\mu}) & \langle v \mid \tilde{\mu}x.c \rangle \rightarrow_{\overline{\lambda}\mu\tilde{\mu}} c\langle v/x \rangle \end{array}$$

The first of these rules provides the logical reduction for the implication connective, but in a more-general form than simply translating the (β) rule of the λ-calculus. This rule is discussed in detail in [21], but can essentially be seen as a 'breaking-down' of the function application into the evaluation of the body of the function v_1 in the context e, with the term v_2 available to be substituted for x (by the $(\tilde{\mu})$ rule)[9]. Note that the context e is 'cut with' v_1 at an inner level, compared with the argument v_2. By comparing with the usual cut-elimination for the sequent calculus, this corresponds to one of the two bracketings of the usual logical reduction rule for implication. The dual idea is to allow the argument v_2 to be 'cut with' v_1 first, and then insert the result into the context e. This corresponds to a different bracketing of the two resulting cuts, and is less naturally expressible in the syntax of $\overline{\lambda}\mu\tilde{\mu}$ because of the lack of an explicit name for the *output* of the function $\lambda x.v_1$. In fact, this original cut-elimination step is not simulated by the reductions of $\overline{\lambda}\mu\tilde{\mu}$.

We show that it is possible to encode the $\overline{\lambda}\mu\tilde{\mu}$-calculus into $\nu\lambda\mu$, in such a way that reductions and typings are preserved. The key observation is that, while $\overline{\lambda}\mu\tilde{\mu}$ distinguishes between terms and contexts, in the $\nu\lambda\mu$-calculus there is a construct present to explicitly represent continuations, being the ν-binding. Contexts then, which have a distinguished formula A on the left, representing their input, will be repackaged as continuations (ν-binding the input) of type $\neg A$. Commands, which have no distinguished formula, will correspond to a term of type $\bot$, which can be constructed from a continuation and a term using the $[M]N$ construct. These ideas are made concrete by the following definition.

To simplify the presentation, we extend the alphabet of variable names in $\nu\lambda\mu$ to include Greek letters (which do not, however, have any special meaning in the calculus).

Definition 5.5.17 (Encoding $\overline{\lambda}\mu\tilde{\mu}$)**.** *We encode $\overline{\lambda}\mu\tilde{\mu}$ by the following mapping (which*

[9] This is the rule which inspired us to define the (λ') reduction rule for $\nu\lambda\mu$: see Definition 5.4.6

applies to terms, contexts, and commands alike; one could consider this three mutually-recursive definitions). In the case for encoding a context of the form $v \cdot e$, we assume y to be a fresh variable.

$$\begin{aligned} \overline{\langle v \mid e \rangle} &= [\overline{e}]\overline{v} \\ \overline{x} &= x & \overline{\alpha} &= \alpha \\ \overline{\lambda x.v} &= \lambda x.\overline{v} & \overline{v \cdot e} &= \nu y.[\overline{e}](y\ \overline{v}) \\ \overline{\mu\alpha.c} &= \mu\alpha.\overline{c} & \overline{\tilde{\mu}x.c} &= \nu x.\overline{c} \end{aligned}$$

Note that commands are encoded exactly as continuation applications; the context e takes the role of the continuation whereas the term v is the argument to the continuation. In particular, note that the standard critical pair $\langle \mu\alpha.c_1 \mid \tilde{\mu}x.c_2 \rangle$, which reduces to both $c_1\langle \tilde{\mu}x.c_2/\alpha \rangle$ and $c_2\langle \mu\alpha.c_1/x \rangle$ in the $\overline{\lambda}\mu\tilde{\mu}$-calculus, is encoded as $[\nu x.\overline{c_2}]\mu\alpha.\overline{c_1}$, forming a similar critical pair. We have the following results with respect to this encoding.

Proposition 5.5.18 (Simulation of $\overline{\lambda}\mu\tilde{\mu}$)**.** *1. The mapping $\overline{\cdot}$ is an injection.*

2. *(a) For any command c, if $c : \Gamma \vdash_{\overline{\lambda}\mu\tilde{\mu}} \Delta$ then $\Gamma, \neg\Delta \vdash \overline{c} : \bot$.*
   (b) For any term v, if $\Gamma \vdash_{\overline{\lambda}\mu\tilde{\mu}} v : A \mid \Delta$ then $\Gamma, \neg\Delta \vdash \overline{v} : A$.
   (c) For any context e, if $\Gamma \mid e : A \vdash_{\overline{\lambda}\mu\tilde{\mu}} \Delta$ then $\Gamma, \neg\Delta \vdash \overline{e} : \neg A$.
3. *For any $\overline{\lambda}\mu\tilde{\mu}$ commands c_1,c_2 (or terms, or contexts) , if $c_1 \to_{\overline{\lambda}\mu\tilde{\mu}} c_2$ then $\overline{c_1} \to_{\nu\lambda\mu} \overline{c_2}$.*

Proof. 1. By induction on the definition of the encoding. By inspection of the right-hand sides, the only possibility of a counterexample is if contexts $v \cdot e$ and $\tilde{\mu}x.c$ existed, such that $\overline{v \cdot e} = \nu x.[\overline{e}](x\ \overline{v}) = \nu x.\overline{c} = \overline{\tilde{\mu}x.c}$. This would only be possible if $c = \langle e \mid v' \rangle$ for some term v' such that $\overline{v'} = x\ \overline{v}$. But by inspection, a term is never encoded as an application.

2. By simultaneous induction on the structures of terms, contexts and commands. The only interesting case is for a context $v \cdot e$. By inspection of the type-assignment rule, we must have $A = B \to C$ for some types B and C, with $\Gamma \vdash_{\overline{\lambda}\mu\tilde{\mu}} v : B \mid \Delta$ and $\Gamma \mid e : C \vdash_{\overline{\lambda}\mu\tilde{\mu}} \Delta$. By induction, twice, we obtain $\Gamma, \neg\Delta \vdash \overline{v} : B$ and $\Gamma, \neg\Delta \vdash \overline{e} : \neg C$. The derivation of Figure 5.2 completes the case.

3. We show here only the case for the $(\to')$ rule (the other cases being simpler). We

$$\dfrac{\dfrac{\Gamma, \neg\Delta, y : B{\to}C \vdash \overline{e} : \neg C \qquad \dfrac{\dfrac{}{\Gamma, \neg\Delta, y : B{\to}C \vdash y : B{\to}C}\,(Ax) \qquad \Gamma, \neg\Delta, y : B{\to}C \vdash \overline{v} : B}{\Gamma, \neg\Delta, y : B{\to}C \vdash y\,\overline{v} : C}\,(\to\mathcal{E})}{\Gamma, \neg\Delta, y : B{\to}C \vdash [\overline{e}](y\,\overline{v}) : \bot}\,(\neg\mathcal{E})}{\Gamma, \neg\Delta \vdash \nu y.[\overline{e}](y\,\overline{v}) : \neg(B{\to}C)}\,(\neg\mathcal{I})$$

Figure 5.2: Derivation for proof of Proposition 5.5.18

have:

$$\begin{array}{rcll}
\overline{\langle \lambda x.v_1 \mid v_2 \cdot e\rangle} & = & [\overline{v_2 \cdot e}]\overline{\lambda x.v_1} & \\
& = & [\nu y.[\overline{e}](y\ \overline{v_2})]\lambda x.\overline{v_1} & \\
& \to_{\overline{\lambda}\mu\tilde{\mu}} & [\overline{e}]((\lambda x.\overline{v_1})\ \overline{v_2}) & (\nu) \\
& \to_{\overline{\lambda}\mu\tilde{\mu}} & [\overline{e}]\mu z.[\nu x.[z]\overline{v_1}]\overline{v_2} & (\lambda') \\
& \to_{\overline{\lambda}\mu\tilde{\mu}} & [\nu x.[\overline{e}]\overline{v_1}]\overline{v_2} & (\mu\neg_2) \\
& = & [\overline{\tilde{\mu}x.\langle v_1 \mid e\rangle}]\overline{v_2} & \\
& = & \overline{\langle v_2 \mid \tilde{\mu}x.\langle v_1 \mid e\rangle\rangle} &
\end{array}$$

□

In [21], the following remark is made: "Without logical or computational loss, one may force the body of a λ-abstraction to have the form $\mu\alpha.c$ (expanding $\lambda x.v$ as $\lambda x.\mu\alpha.\langle v \mid \alpha\rangle$ when necessary)". We observe that if this approach were to be taken, then the rule $(\to')$ could be replaced by the following rule:

$$(\to'')\ \langle \lambda x.\mu\alpha.c \mid v \cdot e\rangle \to_{\overline{\lambda}\mu\tilde{\mu}} \left\{ \begin{array}{c} \langle v \mid \tilde{\mu}x.\langle \mu\alpha.c \mid e\rangle\rangle \\ \textit{or} \\ \langle \mu\alpha.\langle v \mid \tilde{\mu}x.c\rangle \mid e\rangle \end{array} \right\}$$

In this way, the correspondence with the logical cut elimination rule for implication would be restored. However, we note further that this rule is already fully simulated in the $\nu\lambda\mu$-calculus. The first alternative is reachable as demonstrated in the proof above, but the second can also be achieved as follows:

$$
\begin{array}{rcll}
\overline{\langle \lambda x.\mu\alpha.c \mid v \cdot e \rangle} & = & [\overline{v \cdot e}]\overline{\lambda x.\mu\alpha.c} & \\
& = & [\nu y.[\overline{e}](y\ \overline{v})]\lambda x.\mu\alpha.\overline{c} & \\
& \rightarrow_{\overline{\lambda}\mu\tilde{\mu}} & [\overline{e}]((\lambda x.\mu\alpha.\overline{c})\ \overline{v}) & (\nu) \\
& \rightarrow_{\overline{\lambda}\mu\tilde{\mu}} & [\overline{e}]\mu z.[\nu x.[z]\mu\alpha.\overline{c}]\overline{v} & (\lambda') \\
& \rightarrow_{\overline{\lambda}\mu\tilde{\mu}} & [\overline{e}]\mu z.[\nu x.\overline{c}\langle z/\alpha\rangle]\overline{v} & (\mu\neg_2) \\
& = & [\overline{e}]\mu\alpha.[\nu x.\overline{c}]\overline{v} & (\alpha \textit{ conversion}) \\
& = & [\overline{e}]\mu\alpha.\overline{\langle v \mid \tilde{\mu}x.c\rangle} & \\
& = & \overline{\langle \mu\alpha.\langle v \mid \tilde{\mu}x.c\rangle \mid e\rangle} &
\end{array}
$$

The reductions in $\nu\lambda\mu$ are therefore powerful enough to simulate a stronger notion of reduction than that already present in $\overline{\lambda}\mu\tilde{\mu}$. We will expand on this result in Chapter 7, which further relates reduction in classical sequent calculus with reduction in classical natural deduction.

5.6 Thoughts on Strong Normalisation

For any term calculus based on a Curry-Howard correspondence, we regard the strong normalisation of typeable terms to be an essential property. This result guarantees that, in the subset of the language corresponding to *proofs* (the typeable terms), the reduction rules specify a set of proof reductions which are guaranteed to terminate. Unfortunately, although we conjecture the $\nu\lambda\mu$-calculus to be strongly normalising, we have not yet managed a proof of this result. We consider here the main technical difficulties with constructing such a proof.

In the context of the non-constructive behaviour associated with classical logic, the computability techniques of Tait [90] and Girard [41] (a powerful tool for proving strong normalisation for other calculi) are not directly applicable. This is explained by Barbanera and Bernard in [11], in which they illustrate that the intuitive notion of computability for their calculus (also based on classical logic) creates a circularity. Adapting their explanation to the case of $\nu\lambda\mu$, the intuitive definitions would include:

1. $\nu x.M$ is computable if, for all terms N with the same type as x, $M\langle N/x\rangle$ is computable.

2. $\mu x.M$ is computable if, for all terms N with the same type as x, $M\langle N/x\rangle$ is computable.

This immediately creates a circularity in the definitions: the computable terms of type $\neg A$ depend on those of type A, and vice versa. Barbanera and Berardi instead introduce a generalisation of the reducibility technique, using *stratified candidates*, from which the desired candidate sets can be built using a fix-point operation (see [11] for a very clear proof). Their technique has been used by Urban [92], and also by Yamagata [106] to prove the strong normalisation of Parigot's Symmetric $\lambda\mu$-calculus [70] and the extension of the Symmetric λ-calculus to second order logic [107].

Since our $\nu\lambda\mu$-calculus is similar in some respects to the Symmetric $\lambda\mu$-calculus, it seems that this technique ought to be useful in proving the strong normalisation of $\nu\lambda\mu$. One obstacle to the application of this technique is that we have allowed $\bot$ to be a proper type, and negation to be a standard logical connective, in-keeping with the original logic, but in contrast to the approaches of [11] and [70]. It appears that the extension to allow negation to be a proper connective should be possible. Having $\bot$ as a type creates more difficulties, but we might be satisfied with dropping this aspect of our calculus, in order to obtain a strong normalisation result.

An alternative approach, and one which we had hoped to successfully apply, would be to encode the $\nu\lambda\mu$-calculus into the $\mathcal{X}^i$-calculus, for which a strong normalisation result is already known. Unfortunately, for the reasons outlined in Chapter 7, this has not been possible; essentially the reductions in $\nu\lambda\mu$ include behaviour which is not easily simulatable in the $\mathcal{X}^i$-calculus.

5.7 Further Related Work

Streicher and Russ [88] define a variant of $\lambda\mu$-calculus, in which the syntax of $\lambda\mu$ is extended as follows: in the place of α in the syntax construct $[\alpha]M$, they allow *lists* of terms, terminated by a μ-variable (a Greek variable). This extra flexibility allows for "...a considerable simplification of the equational presentation of $\lambda\mu$-calculus." It can be seen to be a step in the same direction as our work, since their lists of terms of the form $M_1 :: M_2 :: \ldots :: M_n :: \alpha$ could be represented in the $\nu\lambda\mu$-calculus by the explicit continuation term $\nu z.[\alpha](((z\ M_1)\ M_2) \ldots M_n)$.

A CPS translation of the $\lambda\mu$-calculus is presented by de Groote [26], which allows a clear understanding of the control behaviour present in the calculus. It would be interesting to show how to extend this work to confluent subsystems of the $\nu\lambda\mu$-calculus, and even to examine which terms (if any) are interpreted in the same way under all CPS translations; in this way we could highlight the non-confluent aspects of the syntax in a clearer manner.

A variant of the $\lambda\mu$-calculus has been used as the basis of a calculus to study the simultaneous inclusion of control features and state (assignable references) by Støvying and Lasses [87]. Their paper is principally concerned with the definition of a syntactic theory based on call-by-value reduction and incorporating these two features. In their formulation, μ-reductions are defined using a "substitution" of (named) evaluation contexts for μ-variables, which essentially behaves in the same way as our semi-structural substitution.

The $\lambda\Delta$-calculus of Rehab and Søresend [77] is also based closely on a Gentzen-style classical logic (indeed, this was the motivation for their work). However, their notion of reduction is relatively weak: the reductions of the calculus are subsumed by (variants of) those of the $\lambda\mu$-calculus, which we have further generalised in the definition of $\nu\lambda\mu$.

The closest work we have seen to the $\nu\lambda\mu$-calculus is that of Rocheteau [81]. He identifies the collection of $\lambda\mu$ reductions existing in the literature, and sets out to define a generalisation of $\lambda\mu$ which can both encompass them, and be compared directly with the $\overline{\lambda}\mu\tilde{\mu}$-calculus. As such, he has very similar aims to our work. However, he chooses to introduce a second class of syntax for 'contexts', whose typing rules correspond to left-introduction rules from the sequent calculus. In order to combine the terms with the contexts, he employs a rule similar to the cut rule of the sequent calculus. Thus, contrary to our stated aim, he does not inhabit Gentzen's classical natural deduction.

5.8 Summary

We have defined a new programming calculus, based essentially on the well-known $\lambda\mu$-calculus, but with various extensions and modifications made in order to obtain a natural Curry-Howard correspondence with a system of classical natural deduction close to Gentzen's original. We have arrived at a set of reduction rules which are both (intentionally) non-confluent, and general enough both to subsume most of those employed in other variants of the $\lambda\mu$-calculus, and to allow the simulation of a fairly general cut-elimination procedure for classical sequent calculus (i.e., the reductions of the $\overline{\lambda}\mu\tilde{\mu}$-calculus).

A natural question to ask of this work is, why has it not been done before? Although it is not yet possible to be sure that the calculus we have presented is strongly-normalising, we believe this to be the case, and if so, the $\nu\lambda\mu$-calculus could be considered a well-behaved basis on which to build other calculi, or to study confluent subsystems. Furthermore, it is pleasing from a philosophical point of view that the calculus achieves a correspondence in the 'spirit' of the original Curry-Howard correspondence; the match between the $\nu\lambda\mu$-calculus and a standard presentation of classical natural deduction is just as clean as that between λ-calculus and minimal natural deduction. It seems that the real answer to this

question is that the need to achieve confluence in applicative-style calculi appears to have been dominant, even in the study of such calculi based on classical logic. As we have argued previously, we do not believe confluence to be a natural feature of calculi in this area, and while it can always be achieved by sufficiently restricting reductions, we believe it is valuable to first explore the problem in general. This approach does (on the other hand) seem to have been common in the more-recent study of calculi based on classical sequent calculus, perhaps because a well-understood notion of reduction was already established in this paradigm (being cut elimination), which is itself naturally non-confluent. Gentzen, of course, was not concerned with matters such as confluence (or even strong normalisation), since for him the value in defining a cut elimination procedure was in proving the *existence* of a cut-free proof. Since the notion of cut-elimination is so well established, we regard the ability of the $\nu\lambda\mu$-calculus to encode these reductions to be a key result. In Chapter 7 we will return to this point and show how to encode the $\mathcal{X}^i$-calculus (whose reductions are still more general than those of $\overline{\lambda}\mu\tilde{\mu}$) into $\nu\lambda\mu$.

It is also important to consider how this calculus, with its abstract and largely mathematical origins, can be related to 'real' programming. The most common parallel which is drawn here is to compare calculi based on classical logics with functional languages extended with control operators. An analysis of this question, in the context of the $\nu\lambda\mu$-calculus, is the subject of the next chapter.

Chapter 6

Control Operators

6.1 Overview

Control operators are syntactic constructs added to functional calculi (and related languages) whose reduction behaviours give explicit control over the *context* in which an expression is evaluated (sometimes also referred to as its *continuation*). It was the seminal paper of Griffin [43] which first sparked an interest in the relationship between functional calculi with control operators and classical logic. Indeed, this was the point at which researchers began to seriously consider the notion of a Curry-Howard correspondence for classical logic.

In Griffin's paper, he observes that the control operator $\mathcal{C}$ can, in certain situations, be assigned the type $((A{\to}\bot){\to}\bot){\to}A$, i.e., its type can be that of the double-negation-elimination axiom of classical logic. The suggestion that programming constructs might inhabit classical types was a striking one, since up to that point the prevalent view was that it was only *constructive* logic which could have a computational content. The early work of Murthy, based on Griffin's initial observation, made more explicit the possibility of viewing classical proofs as programs [61]. Since then, a wealth of research has sprung up investigating possible correspondences between classical logics and computation.

In this chapter, we relate the $\nu\lambda\mu$-calculus to the practical area of *control operators*. We consider a number of well-known control operators from the literature, and explore the idea that they may be represented by the canonical $\nu\lambda\mu$-terms of the appropriate types. In many cases, we find a close correspondence between such $\nu\lambda\mu$-terms and the behaviours of the original operators. We show that the μ-binding of the $\nu\lambda\mu$-calculus can be seen to add a notion of *delimited control*, as provided by (for example), Felleisen's $\mathcal{F}$ operator [37, 38]. As a consequence of this approach, we also discover that a variant of

Felleisen's $\mathcal{F}$ operator corresponds most closely with the canonical inhabitant of double-negation-elimination, and is a better candidate than $\mathcal{C}$ for a control operator corresponding to double-negation-elimination in the logic.

6.2 Existing Control Operators

In this section we examine possible representations for various control operators from the literature in the $\nu\lambda\mu$-calculus. In particular, we will examine the operators $\mathcal{C}$, $\mathcal{A}$, $\mathcal{F}$ and $\#$ as introduced by Felleisen [37, 38] (which he dubs the 'Indiana Control Operators' in [36]). We show that the representations of these operators can be deduced from their type: given a control operator with a type A, the $\nu\lambda\mu$ equivalent can be found by inhabiting the simplest proof of the formula A.

In order to give a concise description of the behaviour of these various operators we need to introduce some more-specific notions of context. Firstly, since most control operators are defined using *applicative contexts*, we give a general definition of these. We allow ourselves the slight liberty of applying this definition to any applicative calculus (e.g., we will use it for other variants of λ-calculus in this chapter). We also define a slight extension of these contexts to the case of an applicative context surrounded by a continuation application (c.f. Definition 5.3.1), since this will be useful for characterising μ-reductions later on.

Definition 6.2.1 (Applicative contexts)**.** *We define applicative contexts C_a (with a unique hole, $\bullet$)*[1] *as follows:*

$$\begin{aligned} C_a ::= {} & \bullet \\ | \;\; & C_a\ M \\ | \;\; & M\ C_a \end{aligned}$$

We define continuation-delimited applicative contexts C_c *by the following grammar:*

$$\begin{aligned} C_c ::= {} & [C_a]M \\ | \;\; & [M]C_a \end{aligned}$$

We will sometimes wish to informally discuss type-assignment for terms described using these contexts. To do this, we employ the following rule (which is easily shown to be admissible), which is essentially a standard rule for typing substitution (in this case,

[1] In the literature, contexts are often alternatively written with [] to denote their 'hole', e.g., $C[\bullet]$ instead of $C_a\{\bullet\}$. We choose our notational variation to avoid confusing with the continuation applications in our calculus (terms of the form $[M]N$).

of a term into the 'hole' of the context). We also allow the corresponding rule with continuation-delimited contexts C_c instead of applicative contexts C_a

$$\frac{\Gamma \vdash M : A \quad \Gamma, z : A \vdash C_a\{z\} : B}{\Gamma \vdash C_c\{M\} : B}\,(\mathit{context})^{(z \notin C_a\{\bullet\})}$$

6.2.1 Undelimited Control Operators

Abort

The simplest control operator in the literature was invented by Felleisen, and presented in [37]. It is called 'abort', and represented as $\mathcal{A}$. It provides the crudest facility for directly manipulating the outlying context; when applied to an argument M in an applicative context C_a, the context is abandoned (deleted), and M is the result. This behaviour is described by the following reduction rule:

Definition 6.2.2 (The abort operator)**.** *The behaviour of abort ($\mathcal{A}$) is described by the following reduction rule:*

$$C_a\{\mathcal{A}(M)\} \rightarrow M$$

For example, the program $(x\ (\mathcal{A}(y)\ z))$ can be reduced by this rule to y. The problem with the reduction rule above, is that it is obviously not compatible; if the redex is placed within a further context, then the new behaviour is to abandon this context also. More explicitly, $C_a\{\mathcal{A}(M)\} \rightarrow M$ but $C'_a\{C_a\{\mathcal{A}(M)\}\}$ does not in general reduce to $C'_a\{M\}$. The original solution to this problem was to allow the rule to be applied only when the context is the 'entire program'. However, as Felleisen discussed in [36], this is not an elegant solution; a better approach is to deal with a delimited version of this operator, as we shall discuss later (subsection 6.2.3).

Call/cc

Probably the most well-known control operator is 'call-with-current-continuation', or 'call/cc' for short. Not only is it widely referred to in the literature, but it is implemented in several popular languages. Scheme [1] is the classic example of a language including call/cc, but a variant of the operator is also included in the Standard ML of New Jersey [59].

Like many of the operators discussed in this chapter, call/cc provides direct access to the surrounding *context* or *continuation* at the point of invocation (the 'current continuation').

The behaviour of the operator is to make a copy of this context and reify it as a special kind of function (see the definition below). If this function is called with an argument, the effect is to throw back that argument to the copy of the context, and to abort whatever other evaluation was currently taking place. For this reason, the kind of language construct created by call/cc is sometimes referred to as an *abortive continuation*.

We will use the symbol $\mathcal{K}$ to represent the call/cc operator in a functional language, and write $\mathcal{K}(M)$ for the application of call/cc to an argument M. The behaviour of this operator can be described by a simple operational rule, in terms of the previously-discussed 'abort' operator.

Definition 6.2.3 (The call/cc operator)**.** *The behaviour of call/cc ($\mathcal{K}$) is described by the following reduction rule:*

$$C_a\{\mathcal{K}(M)\} \rightarrow C_a\{M\ (\lambda x.\mathcal{A}(C_a\{x\}))\}$$

We can see that the behaviour of the operator is to build an abstraction of the current context, with an additional 'abort' inserted around the copy of the context. This kind of term is referred to in the literature as an *abortive continuation*; when an argument is passed to it, the argument is evaluated in the copy of the context, and the surrounding 'abort' abandons any other evaluation still to take place. In order to better illustrate the behaviour of this operator, we consider examples of the reduction behaviour of terms of the form $C_a\{\mathcal{K}(\lambda y.M)\}$. Firstly, if M does not make use of the reified abortive continuation, then evaluation proceeds as normal. Concretely, if $y \notin M$, we have $C_a\{\mathcal{K}(\lambda y.M)\} \rightarrow C_a\{(\lambda y.M)\ (\lambda x.\mathcal{A}(C_a\{x\}))\} \rightarrow C_a\{M\}$. On the other hand, suppose the abortive continuation (inserted for y) is applied to an argument (M') within the body of M. For simplicity, we assume in this case that there is a unique occurrence of y in M, and that M can be written in the form $C'_a\{y\ M'\}$ (in general this may not be possible; for example the application may occur underneath a λ-abstraction). In this case, we observe the following behaviour:

$$\begin{aligned} C_a\{\mathcal{K}(\lambda y.C'_a\{y\ M'\})\} &\rightarrow C_a\{(\lambda y.C'_a\{y\ M'\})\ (\lambda x.\mathcal{A}(C_a\{x\}))\} \\ &\rightarrow C_a\{C'_a\{(\lambda x.\mathcal{A}(C_a\{x\}))\ M'\}\} \\ &\rightarrow C_a\{C'_a\{\mathcal{A}(C_a\{M'\})\}\} \\ &\rightarrow C_a\{M'\} \end{aligned}$$

This example shows that call/cc provides the facility to abandon evaluation of M and return an answer M' to the original context C_a. The 'abort' in the reduction rule for $\mathcal{K}$ is crucial for achieving this behaviour, for two reasons. Firstly, it is the abort operator which allows the remainder of M (being the context C'_a) to be discarded. Secondly, since the

outer context C_a is copied in the rule, the 'abort' ensures that the context is not evaluated twice, by discarding one copy.

Note that in general, there could be multiple occurrences of y in M, and so the kind of reduction described above would not obviously be deterministic. If confluence is a requirement, this is usually resolved by a more-restrictive notion of *evaluation context*, which ensures a unique decomposition of a term into a redex and evaluation context.

'Control'

The operator 'control' (which we write as $\mathcal{C}$), was also presented by Felleisen in [37]. It provides a behaviour similar in some ways to that of $\mathcal{K}$, but with an important difference: the outer context is *not* copied by $\mathcal{C}$, but is only captured within the reified abortive continuation. This behaviour is explicitly described by the following definition:

Definition 6.2.4 (The $\mathcal{C}$ operator)**.** *The behaviour of 'control' ($\mathcal{C}$) is described by the following reduction rule:*

$$C_a\{\mathcal{C}(M)\} \rightarrow M\ (\lambda x.\mathcal{A}(C_a\{x\}))$$

The $\mathcal{C}$ operator gives its argument more-complete control over its surrounding context than $\mathcal{K}$ does. As a special case it is possible to completely discard the context, i.e., $\mathcal{A}$ can be simulated. This can be achieved as follows:

$$\mathcal{A}_{\mathcal{C}}(M) =_{def} \mathcal{C}(\lambda k.M) \qquad (k \notin M)$$

The $\mathcal{C}$ operator can also simulate $\mathcal{K}$, by explicitly reintroducing the copying of the bound context:

$$\mathcal{K}_{\mathcal{C}}(M) =_{def} \mathcal{C}(\lambda k.(k\ (M\ k)))$$

Conversely, $\mathcal{K}$ alone cannot express $\mathcal{C}$ (as it is unable to express the required $\mathcal{A}$ step) [86].

Remark 6.2.5. *Recall that we argued that the two reasons for the inclusion of $\mathcal{A}$ in the reduction rule for $\mathcal{K}$ were: firstly, to avoid evaluating both copies made of the surrounding context, and secondly, to allow the evaluation of the argument to be 'aborted'. In the case of the $\mathcal{C}$ operator, it is not so clear that $\mathcal{A}$ is necessary. Firstly, since the outer context C is not copied, there is no need to discard one copy. Secondly, since $\mathcal{C}$ can directly express $\mathcal{A}$*

(and the way it is expressed depends only on the way in which the context is captures, not on the extra $\mathcal{A}$ introduced by the reduction rule for $\mathcal{C}$), this $\mathcal{A}$ could be inserted manually with another occurrence of $\mathcal{C}$. These points seem to suggest that the inclusion of $\mathcal{A}$ in the reduction rule is somewhat redundant as far as expressibility is concerned. In fact, Felleisen later introduced an operator $\mathcal{F}$ which has a similar behaviour to $\mathcal{C}$, but does not include $\mathcal{A}$ in its reduction rule. The removal of $\mathcal{A}$ actually leads to better properties, as we shall shortly discuss.

6.2.2 Type Assignment

It was Griffin's seminal paper [43] which first sparked interest in the search for an extension of the Curry-Howard Correspondence to a classical logic. His observation was that it is possible to type the $\mathcal{C}$ operator with the type $\neg\neg A \to A$ (for any type A), suggesting that a logical correspondence could be found with the usual implicative logic extended with double-negation elimination (yielding a classical logic). In fact, since there is no explicit negation present in Griffin's work, his proposed typing for the $\mathcal{C}$ operator was $((A \to \bot) \to \bot) \to A$ for any type A.

Considering the reduction rules presented above, one can derive the most general consistent typing for the operators. For example, in the case of 'abort', say M has type B, and the hole in C_a has type A. Then $\mathcal{A}$ must be typed as $B \to A$, for any types A and B. From a logical perspective, this is inconsistent, and Griffin proposed to insist that B be the special type $\bot$, in order to resolve the inconsistency; under this interpretation, $\mathcal{A}$ has the type $\bot \to A$, which is a tautology (of intuitionistic and classical logic, but not minimal logic). This suggests that $\mathcal{A}$ is the computational counterpart of the $(\bot\mathcal{E})$ of natural deduction. For the purpose of these discussions, we will therefore allow the following type-assignment rule:

$$\frac{\Gamma \vdash M : \bot}{\Gamma \vdash \mathcal{A}(M) : A}\,(\bot\mathcal{E})$$

Considering the most-general typing for $\mathcal{K}$, we recall the reduction rule:

$$C_a\{\mathcal{K}(M)\} \to C_a\{M\ (\lambda x.\mathcal{A}(C_a\{x\}))\}$$

Suppose that C_a has a hole of type A, and the 'output type' of C_a is some type D (i.e., when a term of type A is inserted in the hole of the context C_a, the resulting term has type

D). Suppose also that M has some type C. Then, from the left-hand side of the rule, $\mathcal{K}$ must be given type $C{\rightarrow}A$:

$$\dfrac{\dfrac{\Gamma \vdash \mathcal{K} : C{\rightarrow}A \quad \Gamma \vdash M : C}{\Gamma \vdash \mathcal{K}(M) : A}\,(\rightarrow\mathcal{E}) \qquad \Gamma, z : A \vdash C_a\{z\} : D}{\Gamma \vdash C\{\mathcal{K}(M)\} : D}\,(\textit{context})$$

Considering the right-hand side of the reduction rule, we see that the variable x (placed in the 'hole' of C_a) must have type A. $C_a\{x\}$ must have type $\bot$ (according to our typing for 'abort'), i.e., $D = \bot$, and then $\mathcal{A}(C_a\{x\})$ can be assigned any type B, say, as shown:

$$\dfrac{\dfrac{\dfrac{}{\Gamma, x : A \vdash x : A}\,(\textit{ax}) \qquad \Gamma, x : A, z : A \vdash C_a\{z\} : \bot}{\Gamma, x : A \vdash C_a\{x\} : \bot}\,(\textit{context})}{\Gamma, x : A \vdash \mathcal{A}(C_a\{x\}) : B}\,(\bot\mathcal{E})$$

The term $\lambda x.\mathcal{A}(C_a\{x\})$ then has type $A{\rightarrow}B$, meaning that M must have type $(A{\rightarrow}B){\rightarrow}A$, i.e., we require $C = (A{\rightarrow}B){\rightarrow}A$. Returning to our typing for $\mathcal{K}$, then, we deduce that the most general consistent type for the operator is $((A{\rightarrow}B){\rightarrow}A){\rightarrow}A$. This type corresponds to Pierce's Law; a formula which is a tautology of classical logic, but not of intuitionistic logic. The 'typed' version of the reduction rule can now be written as follows:

$$\dfrac{\dfrac{\Gamma \vdash \mathcal{K} : ((A{\rightarrow}B){\rightarrow}A){\rightarrow}A \quad \Gamma \vdash M : (A{\rightarrow}B){\rightarrow}A}{\Gamma \vdash \mathcal{K}(M) : A}\,(\rightarrow\mathcal{E}) \qquad \Gamma, z : A \vdash C_a\{z\} : \bot}{\Gamma \vdash C_a\{\mathcal{K}(M)\} : \bot}\,(\textit{context})$$

$$\rightarrow$$

$$\dfrac{\dfrac{\Gamma \vdash M : (A{\rightarrow}B){\rightarrow}A \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : A \vdash x : A}\,(\textit{ax}) \qquad \Gamma, x : A, z : A \vdash C_a\{z\} : \bot}{\Gamma, x : A \vdash C_a\{x\} : \bot}\,(\textit{context})}{\Gamma, x : A \vdash \mathcal{A}(C_a\{x\}) : B}\,(\bot\mathcal{E})}{\Gamma \vdash \lambda x.\mathcal{A}(C_a\{x\}) : A{\rightarrow}B}\,(\rightarrow\mathcal{I})}{\Gamma \vdash M\ (\lambda x.\mathcal{A}(C_a\{x\})) : A}\,(\rightarrow\mathcal{E}) \qquad \Gamma, z : A \vdash C_a\{z\} : \bot}{\Gamma \vdash C_a\{M\ (\lambda x.\mathcal{A}(C_a\{x\}))\} : \bot}\,(\textit{context})$$

Applying the same analysis to the operator $\mathcal{C}$, we obtain $((A{\rightarrow}B){\rightarrow}\bot){\rightarrow}A$ as the most general typing for the operator. This can, in the special case of $B = \bot$, allow $\mathcal{C}$ to be viewed as a computational representation of double-negation elimination, and it was this

point which sparked the interest in the computational content of classical logic. However, although this is a historically-significant observation, as we will argue, there are actually other control operators which are *better* candidates to correspond with the double-negation elimination of classical logic. Note that the type $((A{\to}B){\to}\bot){\to}A$ is in fact logically consistent without identifying B with $\bot$. This more-general typing comes directly from the use of 'abort' in the reduction, which allows one to obtain an arbitrary type (the use of 'abort' is typed $\bot{\to}B$) in the reduction rule.

If one coerces B to be $\bot$, in order to consider 'control' to be the computational counterpart of double-negation elimination, then the occurrence $\mathcal{A}$ in $\mathcal{A}(x)$ is in fact of type $\bot{\to}\bot$, and so seems to be a redundant step from the point of view of the proof. This is discussed by Ariola and Herbelin in [3]: "...these steps are of type $\bot{\to}\bot$. Therefore it seems we have a mismatch. While the aborts are essential in the reduction semantics they are irrelevant in the corresponding proof." They criticise the work of Ong and Stewart [65] and of de Groote [27], since these works do not include the abort steps in the reduction rules for $\mathcal{C}$. The work of de Groote is particularly relevant to our calculus, since he compares the $\lambda\mathcal{C}$-calculus with the $\lambda\mu$-calculus. However, as Ariola and Herbelin observe, in order to obtain a neat correspondence, de Groote in fact adopts the reduction rules related to a different operator of Felleisen's: the $\mathcal{F}$ operator [38], instead of the usual ones for $\mathcal{C}$. We will argue that this is a natural direction to explore; the operator $\mathcal{C}$ is actually not a canonical representation of double-negation elimination, and the behaviour of $\mathcal{F}$ is closer to this goal. We will return to this issue in the next subsection.

Unfortunately, there is still an inconsistency with the general reduction behaviour of the operators above. Take $\mathcal{A}$, for example, and consider a context C_a whose 'hole' is of type A, and, given x of type A, C_a has type B. Then $C_a\{\mathcal{A}(M)\}$ has type A, for any term M of type $\bot$. But this term runs to M, violating subject reduction, unless A happens to be the type $\bot$, also. This is a serious flaw with the correspondence between these operators and their logical counterparts suggested above; any Curry-Howard correspondence should mean that subject reduction is trivially satisfied, since reductions on typeable terms should be valid reductions on proofs. Griffin proposes a solution to this problem by 'wrapping' any program in a top-level context which is guaranteed to be of type $\bot$, and making modifications to the reduction rules to ensure that this special context is never removed during reduction. However, this restriction does not seem very pleasing, from either the computational or the logical point of view, and Felleisen proposed the operator $\mathcal{F}$ to avoid this workaround.

We should emphasise that Griffin's observation was critical for this whole area of research, and the fact that the reduction behaviour for the $\mathcal{C}$ operator (which he did not define himself) does not neatly match up with a perfect Curry-Howard interpretation of

the calculus could not have been avoided by his work. Instead, he gives the 'best fit' possible, between the operator's semantics and a desired type system based on classical logic. However, motivated by the problems described above, Felleisen defined improved versions of the $\mathcal{C}$ operator, whose reduction behaviour did not require these 'fixes' to guarantee a well-behaved system. Similarly, many other authors have proposed comparable well-behaved control operators. The key idea was that, rather than allowing control over the 'whole program', the abstracted context could be *delimited* by a further syntactic entity known as a *prompt*. We will argue that these refined control operators are more-suitable as computational representations of the inference rules of classical logic.

6.2.3 Delimited Control Operators

The $\mathcal{F}$ operator was introduced by Felleisen to 'fix' the problems which the $\mathcal{C}$ operator had (particularly, the undesirable problems with reduction at the top level of a term). He summarises the motivations for introducing this operator in [36]. The behaviour is defined in terms of a second construct $\#$, a *prompt*[2], which delimits the context captured by the effect of $\mathcal{F}$. Although prompts were originally intended to give an explicit representation of the top-level, it was soon realised that they can be added as a first-class syntax construct, and terms of the form $\# M$ as well as $\mathcal{F}\ M$ may occur arbitrarily. We recall the definitions:

Definition 6.2.6 (The $\Lambda_{\mathcal{F}\#}$ calculus [38])**.** *The terms of the $\Lambda_{\mathcal{F}\#}$ calculus are defined by the following grammar:*

$$M, N ::= x \mid \lambda x.M \mid M\ N \mid \mathcal{F}(M) \mid \#(M)$$

The reduction rules are as follows [3]*:*

$$\begin{aligned}
(\lambda x.M)\ N &\to M\langle N/x\rangle \\
(\mathcal{F}\ M)\ N &\to \mathcal{F}\ (\lambda k.(M\ (\lambda m.(k\ (m\ N)))))\\
N\ (\mathcal{F}\ M) &\to \mathcal{F}\ (\lambda k.(M\ (\lambda m.(k\ (N\ m)))))\\
\#\ (\mathcal{F}\ M) &\to \#\ (M\ (\lambda x.x))\\
\#\ V &\to V \quad (\textit{if } V \textit{ is a value})
\end{aligned}$$

[2]The name is chosen for historical reasons: originally the prompt was introduced to be an explicit representation for the top-level of a program, where (in an interactive interpreter) the user types at a prompt!

[3]In fact, we have slightly generalised Felleisen's definition, which allows the third of these rules to be applied only when N is a value (this ensures confluence, but is not a criterion we feel necessary for the purposes of this work, and without it we can allow a uniform treatment of applicative contexts, e.g., for Proposition 6.2.7 below).

The reduction rules for $\mathcal{F}$ in an application (the second and third rules above) are exactly those for the $\mathcal{C}$ operator, but without the abort steps. We observe that the behaviour of $\mathcal{F}$ can in fact be summarised by an operational rule:

Proposition 6.2.7. *For any applicative context C_a (c.f. Definition 6.2.1) and $\Lambda_{\mathcal{F}\#}$ term M, the following reduction is derivable:*

$$C\{\#(C_a\{\mathcal{F}(M)\})\} \to C\{\#(M\ (\lambda x.C_a\{x\}))\}$$

Proof. By induction on the definition of the context C_a.

$C_a = \bullet$ **:** Then for any term N we have $C_a\{N\} = N$. We observe:

$$\begin{aligned} \#(C_a\{\mathcal{F}(M)\}) &= \#(\mathcal{F}(M)) \\ &\to \#(M\ (\lambda x.x)) \\ &= \#(M\ (\lambda x.C_a\{x\})) \end{aligned}$$

$C_a = C'_a\ N$ **:** Then:

$$\begin{aligned} &\#(C_a\{\mathcal{F}(M)\}) = \#(C'_a\{\mathcal{F}(M)\ N\}) \\ &\to \#(C'_a\{\mathcal{F}\ (\lambda k.(M\ (\lambda m.(k\ (m\ N)))))\}) \\ &\to \#((\lambda k.(M\ (\lambda m.(k\ (m\ N)))))\ (\lambda x.C'_a\{x\})) \textit{ (by induction)} \\ &\to \#(M\ (\lambda m.((\lambda x.C'_a\{x\})\ (m\ N)))) \\ &\to \#(M\ (\lambda m.C'_a\{m\ N\})) \\ &= \#(M\ (\lambda m.C_a\{m\})) \end{aligned}$$

$C_a = N\ C'_a$ **:** Similar to the previous case.

□

We consider next the most general way of typing the operator $\mathcal{F}$. The proposition above suggests that subject reduction will require the property that whatever type can be given to $C_a\{\mathcal{F}(M)\}$ can also be given to $M\ (\lambda x.C_a\{x\})$. Suppose then, that C_a has a hole of type A, that $C_a\{x\}$ is of type B (if x is of type A), and that M is of type C. Then the first term forces $\mathcal{F}$ to be of type $C{\to}A$. Considering the second, the variable x must have type A, and so $\lambda x.C_a\{x\}$ has type $A{\to}B$. Since $C_a\{\mathcal{F}(M)\}$ has type B, we require that $M\ (\lambda x.C_a\{x\})$ have type B also. Therefore, M must have type $(A{\to}B){\to}B$, i.e., we require $C = (A{\to}B){\to}B$. This implies that the most-general typing of the operator $\mathcal{F}$ is $((A{\to}B){\to}B){\to}A$. This type can also be seen to represent double-negation elimination,

if B is coerced to be type $\bot$. However, unlike in the case of $\mathcal{C}$, double-negation elimination is the most general logically-consistent typing which we can give to the $\mathcal{F}$ operator. Therefore, considering the types alone, $\mathcal{F}$ is at least as good as $\mathcal{C}$ as an inhabitant of the type $\neg\neg A \to A$.

Note that by giving the type $\neg\neg A \to A$ to the $\mathcal{F}$ operator, we force the term $\lambda x.C_a\{x\}$ to be of type $\neg A$, and the application of M to this term is actually a continuation application. Furthermore, this implies that the prompt $\#$ featuring in these rules is always applied to a term of type $\bot$. The reduction rule $\#(V) \to V$ implies that prompts should be typed as $\bot \to \bot$. Therefore, although there is no problem with this typing in terms of subject reduction, it does impose a restriction on the use of prompts: in Felleisen's work a prompt may be inserted at a point in the program with any type (the type B, in the discussion above), whereas for a logical interpretation, we insist on type $\bot$. However, we are not asserting that this is *the* correct way of typing the $\mathcal{F}$ and $\#$ operators; rather we observe that it is a typing consistent with the reduction rules, and allows us to compare with possible proof reduction rules for a double negation elimination operator. Ariola, Herbelin and Sabry in [7] make a similar observation regarding the typing of prompts from a logical perspective: "...this is a restriction from the point of view of any computationally interesting type system for control for which one would expect the top-level to be of an inhabited type (e.g., the type of integers). But this is really where the Curry-Howard correspondence holds...".

Given these observations about the typing of the various terms above, we would, in the $\nu\lambda\mu$ setting, express the rule characterising $\mathcal{F}$ as follows:

$$C\{\#(C_a\{\mathcal{F}(M)\})\} \to C\{\#([M]\nu x.C_a\{x\})\}$$

This is close to one of the rules we will shortly describe for characterising the μ-reductions of the $\nu\lambda\mu$-calculus (Definition 6.3.1 in the next section):

$$C\{C_c\{\mu x.M\}\} \to C\{M\langle \nu z.C_c\{z\}/x\rangle\}$$

In fact, as we will show, the μ-binder of the $\nu\lambda\mu$-calculus behaves in a similar way to a delimited control operator. In a sense, we have "home-grown" a delimited control operator from our general approach to inhabiting the logic. In order to explore this idea further, we consider the canonical $\nu\lambda\mu$-terms inhabiting $\bot \to A$ (as 'abort' was typed), Pierce's Law (as call/cc), and double-negation-elimination, and examine their computational behaviour.

6.3 Control Operators in $\nu\lambda\mu$

Recall that μ-bound terms propagate outwards through applicative contexts, consuming and binding their context as a continuation. When the level of a continuation application is reached, this behaviour terminates (the μ-binder is removed). We can now give a characterisation of this kind of continued μ reduction by the following operational-style rules:

Lemma 6.3.1 (Characterisation of repeated μ-reductions)**.** *The following reductions are derivable in the $\nu\lambda\mu$-calculus (so long as C_a is not the empty context, $\bullet$):*

$$C_a\{\mu x.M\} \to \mu y.M\langle \nu z.[y]C_a\{z\}/x\rangle$$
$$C_c\{\mu x.M\} \to M\langle \nu z.C_c\{z\}/x\rangle$$

Proof. By straightforward induction on the definition of the contexts, similar to the proof of Proposition 6.2.7. □

We now turn to the inhabitation of the types of interest. Firstly, consider the type $\bot\to A$. To aid comparison with other operators, we find it convenient to consider the inhabitation of a derived rule for $(\bot\mathcal{E})$, which can be simulated by the $(\boldsymbol{PC})$ rule:

$$\frac{\Gamma, x:\neg A \vdash M:\bot}{\vdash \mu x.M : A}\,(\boldsymbol{PC})$$

Therefore, we define $\mathcal{A}_{\nu\lambda\mu}(M) = \mu x.M$ with $x \notin M$. To check this has the correct operational behaviour, we make use of Lemma 6.3.1:

$$\begin{array}{rll} C_a\{\mathcal{A}_{\nu\lambda\mu}(M)\} &= C_a\{\mu x.M\} & (\lambda) \\ &\to \mu y.M\langle \nu z.[y]C_a\{z\}/x\rangle & (\textit{Lemma 6.3.1}) \\ &= \mu y.M & (x \notin M) \end{array}$$

We have not reached M by this reduction, although the context C_a has been discarded, as expected. In fact, the persistence of the μ-binder in the result avoids the problem with subject reduction which is seen with the operational definition of the behaviour of $\mathcal{C}$ (as discussed above). On the other hand, if the context employed were a continuation-delimited one, the full effect of the abort would be seen (the reader may wish to verify that $C_c\{\mathcal{A}_{\nu\lambda\mu}(M)\} \to M$).

Consider next Pierce's Law $(((A\to B)\to A)\to A)$, the type of the call/cc operator (hereafter denoted by $\mathcal{K}$). By seeking the canonical natural deduction proof matching this

type, we obtain the $\nu\lambda\mu$-term $\mathcal{K}_{\nu\lambda\mu} = \lambda x.\mu y.[y](x\ (\lambda z.\mu w.[y]z))$, which inhabits the following derivation (in which terms, and assumptions in non-essential positions, have been omitted).

$$\dfrac{\dfrac{}{\neg A \vdash \neg A}\,(ax) \qquad \dfrac{\dfrac{\dfrac{}{(A{\to}B){\to}A \vdash (A{\to}B){\to}A}\,(ax) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\neg A \vdash \neg A}\,(ax) \qquad \dfrac{}{A \vdash A}\,(ax)}{\neg A, A \vdash \bot}\,(\neg\mathcal{E})}{\neg A, A \vdash B}\,(PC)}{\neg A \vdash A{\to}B}\,(\to\mathcal{I})}{(A{\to}B){\to}A, \neg A \vdash A}\,(\to\mathcal{E})}{(A{\to}B){\to}A, \neg A \vdash \bot}\,(\neg\mathcal{E})}{(A{\to}B){\to}A \vdash A}\,(PC)}{\vdash ((A{\to}B){\to}A){\to}A}\,(\to\mathcal{I})$$

The behaviour of this term in a continuation delimited context can be seen to be that expected of call/cc:

$$\begin{array}{ll}
C_c\{\mathcal{K}_{\nu\lambda\mu}\ M\} & \\
\to C_c\{\mu y.[y](M\ (\lambda z.\mu w.[y]z))\} & (\lambda) \\
\to [\nu x.C_c\{x\}](M\ (\lambda z.\mu w.[\nu x.C_c\{x\}]z)) & (5.2) \\
\to C_c\{M\ (\lambda z.\mu w.[\nu x.C_c\{x\}]z)\} & (\nu) \\
\to C_c\{M\ (\lambda z.\mu w.C_c\{z\})\} & (\nu) \\
= C_c\{M\ (\lambda z.\mathcal{A}_{\nu\lambda\mu}(C_c\{z\}))\} & (\nu)
\end{array}$$

Remark 6.3.2. *Note that this behaviour is seen for a continuation-delimited context C_c: in fact what we have here is a delimited version of the* call/cc *operator, since it only captures the surrounding context up to the next continuation application.*

Recall now, the operational rule defining the action of $\mathcal{C}$:

$$C_a\{\mathcal{C}\ M\} \to M\ (\lambda x.\mathcal{A}(C_a\{x\}))$$

In fact, if the operator is to be related to double-negation elimination, the λ-binder employed here is really building a continuation (a term of negated type); in the $\nu\lambda\mu$-setting we would employ a ν-binder instead. Similarly, the applications of M to this term is in fact a continuation application. Therefore, we would represent the rule in the richer $\nu\lambda\mu$-syntax as:

$$C_a\{\mathcal{C}\ M\} \rightarrow [M]\nu x.\mathcal{A}(C_a\{x\})$$

In order to try to find an analogous operator in our calculus, we start by finding a natural deduction proof of $\neg\neg A \rightarrow A$. The simplest such proof yields a $\nu\lambda\mu$-term as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x:\neg\neg A, y:\neg A \vdash x:\neg\neg A}\,(ax) \quad \dfrac{}{x:\neg\neg A, y:\neg A \vdash y:\neg A}\,(ax)}{x:\neg\neg A, y:\neg A \vdash [x]y:\bot}\,(\neg\mathcal{E})}{x:\neg\neg A \vdash \mu y.[x]y:A}\,(\boldsymbol{PC})}{\vdash \lambda x.\mu y.[x]y:\neg\neg A \rightarrow A}\,(\rightarrow\mathcal{I})$$

We can show that the term $\lambda x.\mu y.[x]y$ has the following behaviour in a continuation-delimited context:

$$\begin{aligned} C_c\{\mathcal{C}_{\nu\lambda\mu}\ M\} &= C_c\{(\lambda x.\mu y.[x]y)\ M\} \\ &\rightarrow C_c\{\mu y.[M]y\} && (\lambda) \\ &\rightarrow ([M]y)\langle \nu z.C_c\{z\}/y\rangle && (\textit{Definition 6.3.1}) \\ &\rightarrow [M]\nu z.C_c\{z\} \end{aligned}$$

This does not appear to match the behaviour of the $\mathcal{C}$ operator, since no $\mathcal{A}$ steps occur in the redex. However, as suggested previously, these abort steps are not essential for the inhabitation of double-negation-elimination. As we have observed, the $\mathcal{F}$ operator can itself be typed as a double-negation-elimination operator (i.e., given the type $\neg\neg A \rightarrow A$). There is no subject reduction conflict between this typing and the associated reduction rules. Furthermore, the reduction behaviour of the $\nu\lambda\mu$-term $\mathcal{C}_{\nu\lambda\mu}$ above (which we argue is a fairly canonical term inhabiting this type, in a calculus in the style of $\lambda\mu$) is actually closer to $\mathcal{F}$ than $\mathcal{C}$ (there is no 'abort' present around the captured continuation). Instead, we discovered that a $\nu\lambda\mu$-term closely corresponding to the behaviour of $\mathcal{C}$ can be found by taking the canonical inhabitant of the type $\neg(A \rightarrow B) \rightarrow A$; the simplest proof yields a term as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x:\neg(A{\to}B), y:\neg A, z:A, w:\neg B \vdash y:\neg A}(ax) \quad \dfrac{}{x:\neg(A{\to}B), y:\neg A, z:A, w:\neg B \vdash z:A}(ax)}{x:\neg(A{\to}B), y:\neg A, z:A, w:\neg B \vdash [y]z:\bot}(\neg\mathcal{E})}{x:\neg(A{\to}B), y:\neg A, z:A \vdash \mu w.[y]z:B}(PC)}{x:\neg(A{\to}B), y:\neg A \vdash \lambda z.\mu w.[y]z:A{\to}B}(\to\mathcal{I}) \quad \dfrac{}{x:\neg(A{\to}B), y:\neg A \vdash x:\neg(A{\to}B)}(ax)}{x:\neg(A{\to}B), y:\neg A \vdash [x]\lambda z.\mu w.[y]z:\bot}(\neg\mathcal{E})$$
$$\dfrac{\dfrac{x:\neg(A{\to}B), y:\neg A \vdash [x]\lambda z.\mu w.[y]z:\bot}{x:\neg(A{\to}B) \vdash \mu y.[x]\lambda z.\mu w.[y]z:A}(PC)}{\vdash \lambda x.\mu y.[x]\lambda z.\mu w.[y]z:\neg(A{\to}B){\to}A}(\to\mathcal{I})$$

Recalling that our representation of $\mathcal{A}(M)$ in $\nu\lambda\mu$ is $\mu w.M$ where $w \notin M$, it can be seen that the term $\lambda x.\mu y.[x]\lambda z.\mu w.[y]z$ obtained above behaves similarly to the term derived for the $\mathcal{F}$ operator, but with an extra 'abort' step explicitly inserted around the captured copy of the context (which is bound to y). The subterm $\lambda z.\mu w.[y]z$ can be compared with the term $\lambda z.\mathcal{A}(C_A\{z\})$ in the reduction rule for 'control'. So, we naturally obtain a term corresponding closely to $\mathcal{C}$ in our calculus by inhabiting not double-negation-elimination, but the type which we have argued is the most general logically consistent type for $\mathcal{C}$.

For these reasons, we regard $\mathcal{F}$ as a *better* candidate than $\mathcal{C}$ to provide a Curry-Howard correspondence with a calculus with double-negation elimination. This explains why, in the works of de Groote [27], and Ong and Stewart [65], it was found that better correspondences could be identified if the abort steps were removed from the reduction rules for $\mathcal{C}$; essentially those authors were employing a version of the $\mathcal{F}$ operator instead. It has been suggested previously that $\mathcal{F}$ might compare better than $\mathcal{C}$ as a computational representation of double-negation elimination [62, 77]. However, as we will shortly show, we believe even $\mathcal{F}$ is not exactly the right control operator for this role.

Remark 6.3.3. *In various calculi in the literature, it is considered an error to use a delimited control operator without a surrounding delimiter (prompt). However, since our reductions are defined locally, there is no problem if a μ-bound term is not enclosed by a continuation application. It will be impossible for the μ-binding to disappear during reduction in this case, but this is not a problem.*

Since the $\nu\lambda\mu$-terms described above exhibit comparable behaviour to delimited versions of the relevant control operators from the literature when placed in *continuation-delimited contexts*, we would like to draw an analogy between these special kinds of context, and the use of prompts in the literature. In fact, there is quite a natural comparison to be made: the context captured by operators such as $\mathcal{F}$ is delimited by the nearest enclosing prompt, whereas the context captured by repeated μ-reductions is delimited by the nearing enclosing continuation application. This suggests that terms of the form $[M]N$ in our calculus, are comparable with terms of the form $\#(M\ N)$ in the style of Felleisen's calculi. In fact,

the practical motivation for including continuation applications separately from function applications in our calculus is exactly this: to delimit the behaviour of the μ-reductions. If we apply the restriction that $\#$ can only be applied to terms of type $\bot$, then these two syntaxes also coincide in their possible typings, so long as the negated type $\neg A$ of M is converted appropriately into an implicative type of the form $A \rightarrow \bot$.

However, the μ-reductions still do not exactly correspond with those of $\mathcal{F}$. In the rule describing $\mathcal{F}$, the prompt remains unaffected on the right-hand side. On the other hand, for $\nu\lambda\mu$, the outer continuation application is absorbed as part of the context C_c. So in fact, our μ operator does not relate to $\mathcal{F}$ precisely either. There are however many other delimited control operators in the literature, which illustrate other possibilities regarding the management of prompts in these reductions. In particular, the choice of whether or not to leave the prompt in its original position, and whether or not to place a prompt around the captured context, yields four subtly-different operators. These are summarised neatly by Kiselyov [53], and by Dybvig et. al. in [33], in which they are described as follows:

$^{-}\mathcal{F}^{-}$ This operator does not leave a prompt in the original position, nor in the captured continuation: $C\{\#(C_a\{^{-}\mathcal{F}^{-}(M)\})\} \rightarrow C\{M\ (\lambda x.C_a\{x\})\}$. This is related to the operator *cupto* of Gunter et. al. [44], and is also called *control0* in [53].

$^{-}\mathcal{F}^{+}$ This operator does not leave a prompt in the original position, but places one around the captured continuation: $C\{\#(C_a\{^{-}\mathcal{F}^{+}(M)\})\} \rightarrow C\{M\ (\lambda x.\#(C_a\{x\}))\}$. This is related to the *spawn* of [48], and is called *shift0* in [53]. It is also briefly referred to by Felleisen, in [36], as $\mathcal{F}^{-}$.

$^{+}\mathcal{F}^{-}$ This operator leaves a prompt in the original position, but not in the captured continuation: $C\{\#(C_a\{^{+}\mathcal{F}^{-}(M)\})\} \rightarrow C\{\#(M\ (\lambda x.C_a\{x\}))\}$. This is Felleisen's $\mathcal{F}$ operator itself.

$^{+}\mathcal{F}^{+}$ This operator leaves a prompt in the original position, and also places one around the captured continuation: $C\{\#(C_a\{^{+}\mathcal{F}^{+}(M)\})\} \rightarrow C\{\#(M\ (\lambda x.\#(C_a\{x\})))\}$. This is Danvy and Filinsky's *shift* operator [25].

Considering the relative expressiveness of these four variants, it is fairly easy to see that a variant which has fewer prompts than a second can always express the second straightforwardly (i.e., extra prompts can easily be added in a translation). Less obviously, it is possible for those with more prompts to simulate those with fewer. This was shown by the work of Shan [85], who showed that *shift* (with the most prompts) can express the other variants, and, more generally, by Kiselyov [53]. We therefore do not regard the distinction between the four variants as essential. However, it is worth pointing out that the

encodings of control operators with fewer prompts are not compositional; a global transformation of an entire program is required in order to simulate the one with the other. Although it is valuable to know exactly how this encoding can be achieved, the fact that it exists is perhaps not all that surprising, since we know that it is always possible to apply a variant of CPS transform to a program in order to obtain a λ-calculus program with the "same content". Therefore, in a sense, all of these paradigms have equal expressiveness, if one allows for global transformations of entire programs. One could instead be concerned with issues of efficiency, and also how naturally a calculus can express a desired operation. However, we do not explore such questions in detail here; we are satisfied that the control behaviour in the $\nu\lambda\mu$-calculus (which we will now describe) is adequate.

We observe that the $\nu\lambda\mu$-term $\lambda x.\mu y.[x]y$ corresponds with the third operator ${}^{-}\mathcal{F}^{+}$ (or *shift0*) above. Since $\lambda x.\mu y.[x]y$ represents the canonical natural deduction proof of double-negation-elimination, and we argue that the $\nu\lambda\mu$-calculus represents a general and canonical computational interpretation of Gentzen's classical natural deduction, we conclude that, rather than the historically-accepted candidate $\mathcal{C}$, the ${}^{-}\mathcal{F}^{+}$ operator arises naturally as the computational counterpart of double-negation elimination. This observation is the main result of this chapter.

6.4 Design Choices in $\nu\lambda\mu$-reductions

Having illustrated the relationship between the μ-reductions of the $\nu\lambda\mu$-calculus and the behaviour of delimited control operators, we can make clearer the reasons for some of the design choices in the reductions of the $\nu\lambda\mu$-calculus.

6.4.1 Negation as a Primitive Connective

Returning to one of the questions of the previous chapter, we can now explain more clearly why it is advantageous to consider negation as a primitive connective, rather than to define it using $\neg A = A{\to}\bot$. In brief, it allows our μ-binding construct to have the behaviour of a delimited control operator (such as $\mathcal{F}$), rather than an undelimited one (such as $\mathcal{C}$). We will elucidate this point here.

We have given an explanation in this chapter of the control behaviour associated with the μ-binding construct in the calculus, and have shown how it can be seen to be a form of delimited control, as is well-studied in the literature on control operators. Although we do not include an explicit 'prompt' operator in the syntax of our calculus, the characterisation of repeated μ-reductions (Lemma 6.3.1) and the definition of continuation-delimited

contexts on which it depends (Definition 6.2.1) show that the control behaviour of a μ-bound term is delimited by the nearest enclosing continuation application (term of the form $[M]N$). This implies that these kinds of application play a role above and beyond that of function application. The reason is that a μ-binder can be removed when the consumed context is of type $\bot$, and it is this chosen behaviour (c.f. rules $(\mu\neg_1)$ and $(\mu\neg_2)$) which causes the construct to behave like a delimited control operator. This is indeed a design choice; one could always construct a new μ-binding oblivious to the fact that the context is of type $\bot$; for example, one could replace $(\mu\neg_1)$ with the rule:

$$(\mu\neg'_1) : [\mu x.M]N \rightarrow \mu y.M\langle\hat{z}([y][z]N)/x\rangle$$

In this case, the μ-binder always persists whenever it is reduced within an application, and we revert to undelimited control. The μ-bound term in the reduct of the rule $(\mu\neg'_1)$ is necessary of type $\bot$, and as previously discussed, it can be seen therefore to be redundant (it seeks a 'continuation' for 'no value'). If the binding is removed, and the variable y replaced by the canonical term of type $\neg\bot$, being $\nu w.w$ (as is employed in the $(\mu\mu)$ and $(\mu\nu)$ rules of the calculus), then the original reduct of the $(\mu\neg_1)$ rule is reached:

$$\begin{aligned} M\langle\hat{z}([y]([z]N))/x\rangle\langle\nu w.w/y\rangle &= M\langle\hat{z}([\nu w.w]([z]N))/x\rangle \\ &\rightarrow M\langle\hat{z}([z]N)/x\rangle\ (\nu) \end{aligned}$$

Therefore, the $(\mu\neg_1)$ rule can be seen as a refinement of the 'obvious' reduction rule. Of course, this point of view depends on the starting point; in fact, since the $\lambda\mu$-calculus was our departure point then it is natural to have μ-binders disappear in such a reduction rule, due to the rule for "μ-renaming" (c.f. Definition 5.2.3):

$$(\mu r) : [\beta]\mu\alpha.M \rightarrow M\langle\beta/\alpha\rangle$$

Therefore, to take another point of view, we have generalised the pattern of $\lambda\mu$ here: μ-bound terms in function applications reduce to μ-bound terms (c.f., rule (μ) of Definition 5.2.3), whereas μ-bound terms in continuation applications can have their μ-bindings removed.

To see most clearly what would happen if one were to treat negation as a defined connective, we can consider just replacing every ν-bound term $\nu x.M$ with a λ-bound one $\lambda x.M$, and every continuation application $[M]N$ with function application $M\ N$. Because we allow $\bot$ to be a proper type in our calculus, there is nothing wrong with the above replacements in terms of the typeability; essentially we are replacing $\neg A$ with $A\rightarrow\bot$, and

replacing continuations with functions returning $\bot$. Furthermore, if one does not make any μ-reductions, then it is fairly clear that the resulting terms behave in similar fashions. The difference then, is in the behaviour of the μ-bound terms themselves. By removing all continuation applications from a term, we remove the delimiting effects these terms have on the μ reductions, which can change the reduction behaviour of the term, by making the μ-reductions 'coarser'.

For example, consider the $\nu\lambda\mu$-term $(\lambda z.x)\ ([\mu w.y]u)$. The μ-bound variable w does not occur in the sub-term y, and so will have the control effect of discarding its context (the term $\mu w.y$ can be read as $\mathcal{A}(y)$). However, this control effect is delimited by the (immediately) surrounding continuation application. Therefore, even if the μ-reductions are chosen to execute first, it is only the variable u which gets discarded, and the eventual result is unchanged:

$$\begin{aligned}(\lambda z.x)\ ([\mu w.y]u) &\to (\lambda z.x)\ y && (\mu\neg_1)\\ &\to \mu v.[\nu z.[v]x]y && (\lambda')\\ &\to \mu v.[v]x && (\nu)\\ &\to x && (\mu\eta)\end{aligned}$$

The reader may wish to verify that x is in fact the only normal form reachable from the original term. However, if we apply the "translation" described above, we obtain the term $(\lambda z.x)\ ((\mu w.y)\ u)$. This can now be reduced in the following way:

$$\begin{aligned}(\lambda z.x)\ ((\mu w.y)\ u) &\to (\lambda z.x)\ (\mu w.y)\ && (\mu\to_1)\\ &\to \mu w.y && (\mu\to_2)\end{aligned}$$

We can see that the μ-bound term is now able to capture more of the context than previously. The inclusion of both continuation and function application as separate constructs in the calculus therefore allows for more control over the reductions: if it is desirable for a μ-bound term to consume the context even beyond a point of type $\bot$, then a function application can always be employed, as shown above. However, the continuation applications themselves allow for the notion of control to be more refined when this is wanted; if they were removed from the calculus then there would be no natural way in which the effect of the μ-reductions could be delimited, and so this potential for refined control behaviour would be lost.

6.4.2 The $(\mu\neg_2)$ Rule

There is another slightly subtle design decision in the $\nu\lambda\mu$-reductions which we can now explain: why our rule $(\mu\neg_2)$ does not in fact correspond directly to the generalisation of the $\lambda\mu$ rule (μr) above (Definition 5.4.6). The 'obvious' generalisation which we could have used was the following rule:

$$(\mu\neg'_2) : [N]\mu x.M \to M\langle N/x\rangle$$

whereas we choose to reduce to the term $M\langle\hat{z}([N]z)/x\rangle$ instead. One reason for this choice is to maintain a symmetry between this rule and the rule $(\mu\neg_1)$; in turn, this makes the uniform treatment of "continuation delimited contexts" possible, as in Definition 6.2.1. However, given the discussions of this chapter, we can now see a practical distinction which can be made, too. If reduction were to follow the pattern of the original $\lambda\mu$ rule, and substitute N for x in M directly, then this would mean that the implicit 'prompt' in the original continuation application would not be retained in the substituted term, resulting in the 'escape' of any control effects within N. For example, consider the term $(\lambda z.x)\ ([\mu w.y]\mu v.(u\ v))$. If we reduce this term according to the standard $\nu\lambda\mu$-reductions, it is ensured that each of the μ-bound terms remains delimited by a continuation application (just as they are surrounded by one in the initial term). For example, it is possible to reduce as follows:

$$\begin{aligned}(\lambda z.x)\ ([\mu w.y]\mu v.(u\ v)) &\to (\lambda z.x)\ (u\ (\nu z.[\mu w.y]z))\ \ (\mu\neg_2)\\ &\to (\lambda z.x)\ (u\ (\nu z.y))\qquad\ \ (\mu\neg_1)\end{aligned}$$

The behaviour of the μ-binder never reaches the outer application, due to the fact that delimiting continuation applications are preserved around the μ-bound terms. However, if one adopts to rule $(\mu\neg'_2)$ instead, then this is no longer the case, as the following reduction shows:

$$\begin{aligned}(\lambda z.x)\ ([\mu w.y]\mu v.(u\ v)) &\to (\lambda z.x)\ (u\ (\mu w.y))\ \ (\mu\neg'_2)\\ &\to (\lambda z.x)\ (\mu w.y)\qquad\ \ (\mu\to'_2)\\ &\to \mu w.y\qquad\qquad\ \ (\mu\to'_2)\end{aligned}$$

In this case, the μ-bound term is allowed to 'escape' and capture the outer context as well. Its control behaviour will only be delimited by another continuation application in some surrounding term. Therefore, in this case, the μ-bound terms would behave more like the *shift* operator (also called ${}^{+}\mathcal{F}^{+}(M)$ in the discussion above); effectively their surrounding prompts are completely discarded when reached. Although there is nothing inherently wrong with this behaviour (indeed, the *shift* operator is well-known in the literature), the fact that this behaviour is not symmetrical with the $(\mu\neg_1)$ rule makes it

undesirable; essentially we would have a control operator which behaves as *shift* if on the right of continuation applications, and like *shift0* if on the left. Furthermore, the ability of the *shift* operator to escape its prompt and find the next dynamically enclosing one makes it difficult to reason about the behaviour of the operator. This would be still worse if such behaviour was context dependent (i.e., sometimes it happens and sometimes it doesn't). For these reasons, and in order to maintain a uniform definition of this control behaviour, we adopt the rule $(\mu\neg_2)$ and maintain the close correspondence with *shift0*.

6.5 Future Work

For the rest of the chapter, we give some informal ideas on how to make the correspondence between $\nu\lambda\mu$ and control operators clearer, and how to extend its ideas.

6.5.1 Formal Correspondences

We consider relating the $\nu\lambda\mu$-calculus with a calculus in the style of $\Lambda_{\mathcal{F}\#}$, but based on the ${}^{-}\mathcal{F}^{+}$ operator. For the moment, we must retain the restriction that prompts can only be applied to terms of type $\bot$, but we will consider the possibility of lifting this restriction in the next subsection.

We consider the following variant of $\Lambda_{\mathcal{F}\#}$ (Definition 6.2.6) which, as in that definition, includes a general non-confluent notion of reduction:

Definition 6.5.1 (The $\Lambda^{-}\mathcal{F}^{+}$ calculus)**.** *The terms of the $\Lambda^{-}\mathcal{F}^{+}$ calculus are defined by the following grammar:*

$$M, N ::= x \mid \lambda x.M \mid M\ N \mid {}^{-}\mathcal{F}^{+}(M) \mid \#(M)$$

The reduction rules are as follows:

$$\begin{aligned}
(\lambda x.M)\ N &\to M\langle N/x\rangle \\
({}^{-}\mathcal{F}^{+}\ (M))\ N &\to {}^{-}\mathcal{F}^{+}\ (\lambda k.(M\ (\lambda j.(k\ (j\ N))))) \\
N\ ({}^{-}\mathcal{F}^{+}\ (M)) &\to {}^{-}\mathcal{F}^{+}\ (\lambda k.(M\ (\lambda j.(k\ (N\ j))))) \\
\#\ ({}^{-}\mathcal{F}^{+}\ (M)) &\to M\ (\lambda x.\#(x)) \\
\#\ V &\to V \quad (\textit{if } V \textit{ is a value})
\end{aligned}$$

This calculus is identical with Felleisen's $\Lambda_{\mathcal{F}\#}$, but for the penultimate rule, which treats the prompt differently (cf. Definition 6.2.6). This difference is exactly what distinguishes

the two control operators: the resulting prompt is attached to the captured continuation, rather than remaining in position around the whole reduction. Unfortunately, we cannot give any formal correspondence results between this calculus and $\nu\lambda\mu$ since, on the one hand, $\nu\lambda\mu$ features ν-binding, whereas $\Lambda^{-}\mathcal{F}^{+}$ has no corresponding construct, and, on the other hand, $\nu\lambda\mu$ reductions feature semi-structural substitution, whereas $\Lambda^{-}\mathcal{F}^{+}$ builds full abstractions for each partially-captured continuation. However, we could envisage a correspondence between a simplified version of $\nu\lambda\mu$ and this calculus. If we replace all ν-bound terms with λ-bound terms in $\nu\lambda\mu$, and modify the (ν) rule to be $[\lambda x.M]N \rightarrow M\langle x/N\rangle$, we obtain a calculus without explicit continuation terms, but which is more suitable for comparison. The key to the comparison is that continuation applications $[M]N$ in $\nu\lambda\mu$ behave as applications with an implicit prompt. For this reason, we can consider mappings between the two calculi by mapping terms of the form $[M]N$ in $\nu\lambda\mu$ into terms of the form $\#(M\ N)$ in $\Lambda^{-}\mathcal{F}^{+}$, and vice versa. Meanwhile, other terms of the form $\#(M)$ in $\Lambda^{-}\mathcal{F}^{+}$ can be interpreted as $[\nu z.z]M$ in $\nu\lambda\mu$, introducing the continuation application to act as the implicit prompt. Note however that a term $[\nu z.z]M$ naturally reduces back to M in $\nu\lambda\mu$, and the 'prompt' delimiting any control effects in M is lost. If we were to define and work with call-by-value restrictions of the two calculi (in which $[\nu z.z]M$ would reduce to M only when M was a value, corresponding with the ${}^{-}\mathcal{F}^{+}(\lambda x.M)$ reduction $\#\ V \rightarrow V$, then this problem could be avoided (at the obvious expense of the full generality of the reductions).

Finally, terms of the form $\mu x.M$ map to terms of the form ${}^{-}\mathcal{F}^{+}(\lambda x.M)$, and terms of the form ${}^{-}\mathcal{F}^{+}(M)$ map back to $\mu x.(M\ x)$. The correspondence between these terms can be understood computationally (for example, the term $\mu x.(M\ x)$, captures and reifies any continuation-delimited context it is placed in, and passes the result as an argument to M, mimicking the behaviour of the ${}^{-}\mathcal{F}^{+}$ operator). It can also be understood logically: they represent the proofs of the inter-derivability of double-negation elimination and proof-by-contradiction, when added to minimal natural deduction. We conjecture that the mappings described above could be used as the basis of equational correspondences between suitably-defined confluent sub-calculi, but this remains future work.

6.5.2 Top Level Types

The restriction to only allow prompts to occur with $\perp$ type seems necessary from the point of view of the logic, but rather restrictive from the programming perspective. In [5, 6], Ariola et. al. explore various type systems for calculi with delimited control which allow more flexible use. In particular, the idea is explored that prompts can occur with different types, even within the same program. The type system needs then to keep track of

the type of the current 'top-level' (i.e., the closest surrounding prompt). This idea works neatly for control operators which retain prompts during reduction (i.e., the variants ${}^{-}\mathcal{F}^{+}$ and ${}^{+}\mathcal{F}^{+}$ above). In the case of the $\nu\lambda\mu$-calculus, this would involve keeping track of the type of the next enclosing $[M]N$ term; these terms would no longer be restricted to have type $\bot$, but could be given a different top-level type. Therefore, the distinction between function application and continuation application would become purely one of delimiting the μ-reductions; both types of application could in principle return any type. For the μ-reductions to work correctly, it would be necessary for the type system to enforce that the type of the surrounding $[M]N$ term agreed with the return type of the μ-bound variable. This kind of extension would be interesting for future work, since it should then be possible for the $\nu\lambda\mu$-calculus to fully simulate an expressive calculus of delimited control, using only its standard syntax. However, the correspondence with the standard presentation of the original logic would of course be lost.

6.6 Summary

We have shown how the $\nu\lambda\mu$-calculus can be understood to be a calculus featuring explicit functions and continuations, plus a notion of delimited control, most closely-related to the *shift0* or *spawn* control operators from the literature. We have argued that this notion of control is sufficient to express most of the other control operators in the literature, although in some cases a direct, compositional encoding is not possible (just as not all existing control operators can express one another in a straightforward manner). Furthermore, although some design decisions have been taken along the way, the reductions of the calculus (and the control behaviour in particular) are essentially "home-grown" from the original logic; we have reached our definitions by applying and generalising existing ideas in (what we hope is) a natural way.

As a result of the logical origins of our calculus, we find that the type of delimited terms (which are continuation applications in our case) is restricted to always be $\bot$. However, as we have remarked, it may be possible to relax this restriction if one is happy to withdraw from the original logical correspondence, and obtain a calculus which may be fully comparable with existing notions of delimited control. A more detailed type system for delimited continuations, allowing extra flexibility, has been presented by Murthy [62].

While it is not particularly surprising that the inclusion of classical negation in the underlying logic is related computationally to adding some notion of control to a functional calculus, we believe that our uniform "first-principles" approach convincingly shows that the particular control behaviour we observe in $\nu\lambda\mu$ is a natural feature of a calculus based

on a Gentzen-style classical logic. What is perhaps most interesting is that we obtain a notion of *delimited* control, without adding any syntactic entities unwarranted from the logical point of view: it is purely the separation of implication from negation (and hence, continuation from function applications) which results naturally in a fairly-general notion of delimited control. In most other works on delimited continuations, these arise out of syntactic constructs added specifically for the purpose. However, in the specific context of call-by-name reduction, Herbelin and Ghilezan have recently observed a similar result for a calculus related to classical logic [47].

Chapter 7

The Relationship Between Cut Elimination and Normalisation

7.1 Overview

So far in this thesis we have worked with the two main logical paradigms of sequent calculus and natural deduction in isolation. In both cases we have presented calculi which we believe to be good candidates to represent the computational content of the logics, and have developed further results based on the definitions. It is natural to consider the relationship between these two calculi and, in the process, the relationship between the reductions specified by cut elimination in the sequent calculus, and by proof normalisation steps in the natural deduction setting. This topic is not new and we shall begin by giving an overview of various work which has been produced in the area. However, to our knowledge there does not exist work in the literature showing how to encode a general non-confluent cut-elimination procedure (as is typical for classical sequent calculus) into the natural deduction paradigm, in such a way that full reduction is preserved. In this chapter we achieve this result by defining an injective encoding of the $\mathcal{X}^i$-calculus (Chapter 3) into the $\nu\lambda\mu$-calculus (Chapter 5), and showing it to preserve reductions and typings. This result advocates further the generality of the reductions of the $\nu\lambda\mu$-calculus.

Having achieved pleasing results concerning the encoding from the sequent calculus paradigm to the natural deduction, it is natural to consider the reverse question, and attempt to reach an encoding in the other direction. In our case, we would like an encoding of the $\nu\lambda\mu$-calculus into $\mathcal{X}^i$, with the same good theoretical properties. Unfortunately, we have not been able to achieve a result of this kind. We give instead a discussion of the difficulties presented. In particular, some of the reductions in $\nu\lambda\mu$ seem less easily

expressible than others, in the sequent calculus paradigm, and we consider what extra reductions these might imply for $\mathcal{X}^i$.

7.2 Previous Encodings Between Sequent Calculus and Natural Deduction

There exist several works in the literature relating sequent calculi with natural deduction proof systems, and studying the corresponding relationships between term calculi based in each of the logical paradigms.

7.2.1 Gentzen's Encodings

The first encodings between sequent calculus and natural deduction were defined by Gentzen in his original publication of these paradigms [39]. He showed that the provability of sequent calculi, natural deduction systems, and Hilbert systems coincide, both for intuitionistic and for classical logic. This was shown by encoding the proofs of each paradigm into the next, so that by composing these encodings, the equal provability of the three systems was established. The main reasons for obtaining these results in Gentzen's work were (presumably) to relate his calculi to the existing notion of deduction defined by Hilbert, and, most importantly, to facilitate the proof of meta-theoretical results, such as consistency of the calculi. In particular, Gentzen was unable to show consistency of his natural deduction calculi directly, but instead, by showing that they were equal to their sequent calculus counterparts in terms of provability, and by showing that the sequent calculi were themselves equivalent to their cut-free fragments (by cut elimination), he could reduce the problem to that of showing consistency of the cut-free fragment. This was then an immediate consequence of the subformula property, which is enjoyed by the cut-free fragments of Gentzen's sequent calculi.

Since Gentzen was only concerned with the existence of corresponding proofs in the other paradigms, he was not especially concerned about the *choice* of encoding employed. For example, there is no necessity for the composition of all three encodings to be the identity map on proofs, or even anything close to this. In addition, Gentzen did not present notions of proof normalisation for the paradigms other than sequent calculus, since the cut elimination result in this setting was sufficient to prove his consistency results. Therefore, no parallel is drawn in this work between notions of proof reduction in the different paradigms.

7.2.2 Prawitz's Encodings

Prawitz includes in his work a short chapter on sequent calculus, in which he gives (implicitly, in the form of an argument) an encoding of natural deduction into sequent calculus which is an improvement on Gentzen's. The significant improvement from the computational point of view is that, in the intuitionistic case, it maps normal natural deduction proofs onto cut-free sequent proofs (this is not the case for Gentzen's encodings). To illustrate this point using term calculi, the λ-calculus term $x\ y$ would be encoded into the $\mathcal{X}^i$-calculus (i.e., the sequent calculus paradigm) as the term $\langle x.\beta\rangle\widehat{\beta} \dagger \widehat{z}(\langle y.\gamma\rangle\widehat{\gamma}\ [z]\ \widehat{w}\langle w.\alpha\rangle)$, in which a cut is introduced. When the cut is eliminated, one reaches the normal form $\langle y.\gamma\rangle\widehat{\gamma}\ [x]\ \widehat{w}\langle w.\alpha\rangle$ which is the result of Prawitz's encoding. The result concerning preservation of normal forms is not stated in his work, however, and no study is made there of the relationship between reductions in these two paradigms.

7.2.3 Urban's Encodings

We believe that the closest work to this thesis, both in terms of content and general philosophy, is Christian Urban's PhD thesis [92]. It is the work of Urban which provides the cut-elimination underpinning for the $\mathcal{X}^i$-calculus, and he is a strong advocate of the point of view that classical logic has a naturally non-confluent set of reductions.

In Urban's work, he includes sections comparing his cut-elimination procedure with a set of reductions for a "natural deduction calculus". At first glance, this would appear to already establish the main result we claim in this chapter, since (in the classical case) his unrestricted cut-elimination is encoded, preserving reductions. However, the "natural deduction calculus" which is employed in Urban's work is not a Gentzen-style calculus (and in fact, we shall argue that it actually bears a closer resemblance to a sequent calculus); it is based on the "sequence-conclusion natural deduction" of Boričić, extended with a 'substitution' rule, and with elimination rules replaced by rules from Parigot's *free deduction* [68]. We recall the definition of Urban's formulation here, in which we omit the rules for conjunction and disjunction, and employ Greek letters for the second alphabet of names, in order to facilitate comparisons with our work.

Definition 7.2.1 (Urban's variant of Boričić sequence-conclusion calculus for classical

logic [92]).

$$\frac{}{\Gamma, x:A \vdash \alpha:A, \Delta}\,(ax) \qquad \frac{\Gamma \vdash \alpha:A, \Delta \quad \Gamma, x:A \vdash \Delta}{\Gamma \vdash \Delta}\,(\textit{Subst})$$

$$\frac{\Gamma, x:A \vdash \alpha:B, \Delta}{\Gamma \vdash \beta:A{\rightarrow}B, \Delta}\,(\rightarrow\mathcal{I}) \qquad \frac{\Gamma \vdash \alpha:A{\rightarrow}B, \Delta \quad \Gamma \vdash \beta:A, \Delta \quad \Gamma, x:B \vdash \Delta}{\Gamma \vdash \Delta}\,(\rightarrow\mathcal{E})$$

$$\frac{\Gamma, x:A \vdash \Delta}{\Gamma \vdash \alpha:\neg A, \Delta}\,(\neg\mathcal{I}) \qquad \frac{\Gamma \vdash \alpha:\neg A, \Delta \quad \Gamma \vdash \beta:A, \Delta}{\Gamma \vdash \Delta}\,(\neg\mathcal{E})$$

The most obvious difference between this presentation and Gentzen's natural deduction is the use of multiple conclusions in the sequents. However, there are other striking similarities with the classical sequent calculus. The existence of the $(\textit{Subst})$ rule essentially equips the calculus with a cut, while the elimination rules (whose forms are usually a distinguishing feature of natural deduction systems) are of a form in which no new conclusion is derived in the consequent. In the case of the $(\neg\mathcal{E})$ rule, this simply means that the occurrence of $\bot$ which is derived as standard, has been removed. Since $\bot$ mainly plays the role of a place-holder for a conclusion, this choice is understandable in a setting with multiple conclusions, and is similar to the situation in sequent calculus, where negation can be included without any need for $\bot$. In the case of implication (and indeed conjunction and disjunction, which we have omitted here), we see that, rather than the traditional Modus Ponens formulation, we have instead *three* premises, the last of which can be seen to 'absorb' the statement B which would usually be derived. This rule, instead of defining the canonical consequences which can be derived from an implicative formula, defines exactly the situation in which the formula can be removed from the sequent, leaving no new conclusion. Therefore, instead of a usual natural deduction elimination rule, what we have here closely resembles the form of a left-introduction rule from the sequent calculus. We observe in fact, that if we take the $(\rightarrow\mathcal{E})$ defined above and restrict the first premise to be an axiom, then we obtain the following form:

$$\frac{\dfrac{}{\Gamma, y:A{\rightarrow}B \vdash \alpha:A{\rightarrow}B, \Delta}\,(ax) \qquad \Gamma \vdash \beta:A, \Delta \quad \Gamma, x:B \vdash \Delta}{\Gamma, y:A{\rightarrow}B \vdash \Delta}\,(\rightarrow\mathcal{E})$$

Removing the axiom entirely would yield exactly the left-introduction rule from the sequent calculus. Therefore, what we have in the calculus above is a generalisation of the left-rule in the sequent calculus. The same observation applies in the cases of conjunction and disjunction. This style of rule is also "generalised elimination rule" and advocated by

von Plato [100].

To summarise, this "natural deduction" presentation has multiple conclusions, a cut rule, and, in place of the standard elimination rules, generalisations of the left-introduction rules from the sequent calculus. Therefore, we argue that it does not truly represent a calculus in the natural deduction paradigm, but is rather a hybrid, closest to the sequent calculus itself. This provides a significant simplification of the problem of obtaining a correspondence with sequent calculus. Urban himself writes that he chooses this formulation in order to simplify the problem: "...the reasons for choosing this particular set of inference rules are entirely pragmatic: they simplify considerably the translations between natural deduction proofs and sequent proofs. Other sets of inference rules can also be used to study the correspondence, although the machinery required is more complicated". We find that the problem is significantly harder in the context of a Gentzen-style presentation of classical natural deduction, and that to present an adequate solution is a non-trivial extension of Urban's correspondence. Indeed, Urban claims "...it is very impractical to do the extension using Gentzen's natural deduction calculus NK, because this calculus requires double negation translations for encoding classical proofs". This suggests that such encodings would not preserve typings (since double negations would be added to the types in various locations), but this is not in fact the case for our work. We achieve a pleasing result in this direction, although our correspondence is only one-way.

7.2.4 The Intuitionistic Case

There have been several publications concerning the relationship between natural deduction and sequent calculus in the intuitionistic case. Gentzen and Prawitz both provided encodings, but did not examine the relationships between the notions of reduction in these settings. The works of Zucker [108] and, later, Pottinger [71], were the first to compare reductions in these two paradigms, in the case of intuitionistic logic. They showed that the two notions of reduction could indeed correspond, but only in a limited fragment of the logics (in particular, the rules for disjunction presented difficulties). However, the correspondence between proofs in sequent calculus and those in natural deduction was many-to-one.

Herbelin [46] presented an improved correspondence by introducing the $\overline{\lambda}$-calculus, which corresponds with a restricted form of intuitionistic sequent calculus. This formulation is the intuitionistic restriction of the work of Joinet et. al. [51, 93], and is called LJT (after their LKT). It forms the basis of work by Dyckhoff and Pinto [34, 35], who show that it provides a useful theoretical bridge between the traditional formalisms of natural deduction and sequent calculus. The key advance in Herbelin's work is that cut-free sequent

proofs are in one-to-one correspondence with normal natural deduction proofs (in contrast to previous attempts). This is achieved by restricting the legal forms of sequent proofs to eliminate the perceived 'redundancy', and employing a λ-calculus extended with explicit substitution and list-construction operators (to explicitly represent a list of arguments, like a call-stack). These works, and others, have been neatly summarised by Barendregt and Ghilezan [13].

Herbelin's work has been more recently extended by Santo [82, 83], who has made progress concerning isomorphisms of reductions, and isomorphisms between non-normal proofs. This work culminated in a recent paper [84] describing an isomorphism between unrestricted intuitionistic sequent calculus and an extension of standard intuitionistic natural deduction, to allow explicit "applicative contexts" in the syntax. He presents a full isomorphism of both terms and reductions. The mappings between the two paradigms reflect Herbelin's original idea of exchanging the associativity of a list of applications. His natural deduction calculus includes "generalised elimination rules".

7.2.5 Other Encodings

Ogata [63] showed that Parigot's $\lambda\mu$-calculus can be encoded into a variant of classical sequent calculus, being the LKT calculus of Danos et. al. [93]. This encoding is shown to preserve reductions, and the nature of the encoding is related to continuation-passing-style transformations. Both the source and target calculi are confluent (and therefore, restricted) calculi based on classical logics.

Curien and Herbelin [21] use translations into the natural deduction paradigm in order to neatly define and motivate the call-by-name and call-by-value restrictions of their $\overline{\lambda}\mu\tilde{\mu}$ calculus. Although their full calculus corresponds with classical sequent calculus, the two translations are to minimal natural deduction (i.e., λ-calculus), which is possible for such confluent restrictions only.

In [102], Wadler relates an extended version of his "dual calculus" to variants of the $\lambda\mu$-calculus. He defines encodings between the dual calculus and $\lambda\mu$ in both directions (separately, for call-by-name and call-by-value versions of each), which preserve equalities in the two calculi. Reductions, however, are not preserved by the encodings, and the encodings for the different evaluation strategies are quite different. Therefore these are not suitable in our setting, where we cannot rely on equational reasoning (due to non-confluence), and wish to provide encodings which preserve reductions (independent of any evaluation strategy).

Audebaud and van Bakel [9] relate the $\lambda\mu$-calculus with the $\mathcal{X}$-calculus [98], giving en-

codings between the two which preserve typings. However, reductions are not directly preserved, rather the weaker property is shown that if a reduction $P \rightarrow Q$ is possible, then the interpretations of the two terms have a common reduct. Similarly, Rocheteau [81] studies the relationship between the $\overline{\lambda}\mu\tilde{\mu}$-calculus [21] and a generalisation of the $\lambda\mu$-calculus, but the simulation results hold only up to 'joinability' in the same way.

7.3 Encoding $\mathcal{X}^i$ into $\nu\lambda\mu$

In order to simplify the presentation of our encoding, we extend the syntax for variables in $\nu\lambda\mu$ to include Greek letters as well as Roman ones. However, we do not consider these to have any special meaning (unlike in $\lambda\mu$); all are considered to be standard term variables in the calculus. We define an encoding $\llbracket . \rrbracket$ in which free occurrences of outputs (plugs) are interpreted as variables of continuation type (a term substituted for the variable can "consume" the output), and bound outputs are interpreted using μ-binders. In order to avoid redundant generation of terms of the form $\mu\alpha.[\alpha]M$ in which α does not occur in M (and thus "smooth" the encoding for our simulation result), we make use of an auxiliary encoding of the form $\llbracket . \rrbracket_\alpha$, which avoids these μ-binders being generated when the input $\mathcal{X}^i$-term already introduces α.

Definition 7.3.1 (Encoding $\mathcal{X}^i$ into $\nu\lambda\mu$)**.**

$$
\begin{aligned}
\llbracket \langle x.\alpha \rangle \rrbracket &= [\alpha]x \\
\llbracket \widehat{x}P\widehat{\alpha}\cdot\beta \rrbracket &= [\beta]\lambda x.\llbracket P \rrbracket_\alpha \\
\llbracket P\widehat{\alpha}\,[x]\,\widehat{y}Q \rrbracket &= [\nu y.\llbracket Q \rrbracket]\big(x\ \llbracket P \rrbracket_\alpha\big) \\
\llbracket \widehat{x}P \cdot \alpha \rrbracket &= [\alpha]\nu x.\llbracket P \rrbracket \\
\llbracket x \cdot P\widehat{\alpha} \rrbracket &= [x]\llbracket P \rrbracket_\alpha \\
\llbracket P\widehat{\alpha} \dagger \widehat{x}Q \rrbracket &= \left\{ \begin{array}{l} \llbracket Q \rrbracket\{\llbracket P \rrbracket_\alpha / x\} \text{ if } Q \text{ introduces } x \\ [\nu x.\llbracket Q \rrbracket]\llbracket P \rrbracket_\alpha \text{ otherwise} \end{array} \right\} \\
\llbracket P \rrbracket_\alpha &= \left\{ \begin{array}{ll} M & \text{if } P \text{ introduces } \alpha, \text{ where } \llbracket P \rrbracket = [\alpha]M \\ \mu\alpha.\llbracket P \rrbracket & \text{otherwise} \end{array} \right\}
\end{aligned}
$$

To be sure that the auxiliary definition $\llbracket P \rrbracket_\alpha$ makes sense, we need the following property:

Proposition 7.3.2. *For all $\mathcal{X}^i$-terms P and plugs α, there exists a $\nu\lambda\mu$-term M such that $\llbracket P \rrbracket = [\alpha]M$ with $\alpha \notin M$, if and only if, one of the following conditions hold:*

1. *P introduces α.*

2. $P = Q\widehat{\beta}\,[x]\,\widehat{y}R$, *and* R *introduces* y, *and there exists a* $\nu\lambda\mu$*-term* N *such that* $[\![Q]\!] = [\alpha]N$ *with* $\alpha \notin N$.

3. $P = Q\widehat{\beta}\,\dagger\,\widehat{y}R$, *and* R *introduces* y, *and there exists a* $\nu\lambda\mu$*-term* N *such that* $[\![Q]\!] = [\alpha]N$ *with* $\alpha \notin N$.

In particular, if P *introduces* α, *then* $[\![P]\!] = [\alpha]M$ *with* $\alpha \notin M$.

Proof. By straightforward induction on the structure of the term P, using Definition 7.3.1. □

The two alternatives when encoding a cut, along with the auxiliary definition $[\![P]\!]_\alpha$ mean that cuts $P\widehat{\alpha}\,\dagger\,\widehat{x}Q$ are potentially encoded in four different ways, depending on whether α and x are introduced. The reason for this non-uniform treatment is that, in the reductions of $\nu\lambda\mu$, binders are not 'reconstructed' when the cut is 'propagated'. So, if we were to encode all cuts as $[\nu x.[\![Q]\!]]\mu\alpha.[\![P]\!]$, our simulation result would not be possible, since terms of this form will not be constructed after reduction.

As an easy first result, we show that the typings possible in the $\mathcal{X}^i$-calculus are preserved by the encoding.

Theorem 7.3.3 (Encoding $\mathcal{X}^i$ to $\nu\lambda\mu$, preserves typings)**.** *For any right-context* Δ, *let* $\Delta = \{\alpha : A \mid \alpha : A \in \Delta\}$. *Then, for any* $\mathcal{X}^i$*-term* P, *we have* $P :\cdot\, \Gamma \vdash \Delta$ *if and only if* $\Gamma, \neg\Delta \vdash [\![P]\!] : \bot$.

Proof. By straightforward induction on the structure of the term P. □

Lemma 7.3.4 (Encoding of cuts)**.** *Let* $P\widehat{\alpha}\,\dagger\,\widehat{x}Q$ *be an arbitrary* $\mathcal{X}^i$*-term of this form (a cut). Then we have:*

1. $\mu\alpha.[\![P]\!] \to [\![P]\!]_\alpha$.

2. $[\nu x.[\![Q]\!]][\![P]\!]_\alpha \to [\![P\widehat{\alpha}\,\dagger\,\widehat{x}Q]\!]$.

3. $[\nu x.[\![Q]\!]]\mu\alpha.P \to [\![P\widehat{\alpha}\,\dagger\,\widehat{x}Q]\!]$.

Proof.

1. In the case where $[\![P]\!]_\alpha = \mu\alpha.[\![P]\!]$, the result holds trivially. On the other hand, if we have $[\![P]\!] = [\alpha]N$, for some term N with $\alpha \notin N$, then $\mu\alpha.[\alpha]N \to [\![P]\!]_\alpha$ by the rule $(\mu\eta)$, as required.

2. If Q does not introduce x, then we are done, since $[\![P\widehat{\alpha} \dagger \widehat{x}Q]\!] = [\nu x.[\![Q]\!][\![P]\!]_\alpha$. Alternatively, if Q introduces x, then $[\![P\widehat{\alpha} \dagger \widehat{x}Q]\!] = [\![Q]\!]\{[\![P]\!]_\alpha/x\}$. We conclude, noting that $[\nu x.[\![Q]\!][\![P]\!]_\alpha \to [\![Q]\!]\{[\![P]\!]_\alpha/x\}$.

3. By combining parts 1 and 2.

□

We observe next that since our encoding introduces a continuation application as part of the encoding of *each* syntactic construct in an $\mathcal{X}^i$-term P, in the special case where P introduces x, we can view the encoding of P as a continuation-delimited context (Definition 6.2.1), in which x marks the 'hole'.

Lemma 7.3.5 (Encoding to Continuation-Delimited Contexts)**.** *If P is an $\mathcal{X}^i$-term such that P introduces x, then there exists a continuation-delimited context C_C such that $C_C\{x\} = [\![P]\!]$.*

Proof. By inspection of the cases $[\![\langle x.\alpha\rangle]\!]$ and $[\![P\widehat{\alpha}\,[x]\,\widehat{y}Q]\!]$ and $[\![x \cdot P\widehat{\alpha}]\!]$ in Definition 7.3.1. □

This allows us to prove a technical lemma, which is useful for dealing with the cases in our encoding where substitutions make unclear the precise structure of the resulting terms.

Corollary 7.3.6. *If P is an $\mathcal{X}^i$-term such that P introduces x, and $\mu y.M$ is a $\nu\lambda\mu$-term, then $[\![P]\!]\langle\mu y.M/x\rangle \to M\langle\hat{x}([\![P]\!])/y\rangle$*

Proof. By combining Lemma 7.3.5 with Lemma 6.3.1. □

The main work towards our simulation result is in relating the meta-operations in the two calculi. Our encoding is defined with the view in mind that right-propagation of cuts should roughly correspond with the usual term substitution in an applicative setting. Left-propagation, on the other hand, can be related to our semi-structural substitutions. This observation appears to be new, and gives an intuitive explanation of what left propagation of cuts might mean in a computational sense: while right-propagation and term substitution bring terms to contexts in which they are used, left-propagation and semi-structural substitution can be used to bring contexts to terms. This relationship (at least one-way) is formalised in the following results:

Lemma 7.3.7 (Simulation of Propagation by Substitutions)**.** *For all $\mathcal{X}^i$-terms P and Q:*

1. *If* $\llbracket Q\{P\widehat{\alpha}\leftrightarrow x\}\rrbracket = \llbracket Q\rrbracket\langle\llbracket P\rrbracket_\alpha/x\rangle$*, and* $\beta \notin fs(P)$ *and* $\alpha \neq \beta$*, then we have* $\llbracket Q\{P\widehat{\alpha}\leftrightarrow x\}\rrbracket_\beta = \llbracket Q\rrbracket_\beta\langle\llbracket P\rrbracket_\alpha/x\rangle$.

2. $\llbracket Q\{P\widehat{\alpha}\leftrightarrow x\}\rrbracket = \llbracket Q\rrbracket\langle\llbracket P\rrbracket_\alpha/x\rangle$.

3. *If* $\beta \neq \alpha$ *and* $\beta \notin fp(Q)$*, then:*

   (a) *If* Q *introduces* x *and* $\llbracket P\{\alpha\leftrightarrow\widehat{x}Q\}\rrbracket = \llbracket P\rrbracket\langle\hat{x}(\llbracket Q\rrbracket)/\alpha\rangle$*, then we have* $\llbracket P\{\alpha\leftrightarrow\widehat{x}Q\}\rrbracket_\beta = \llbracket P\rrbracket_\beta\langle\hat{x}(\llbracket Q\rrbracket)/\alpha\rangle$.

   (b) *If* Q *does not introduce* x *and* $\llbracket P\{\alpha\leftrightarrow\widehat{x}Q\}\rrbracket = \llbracket P\rrbracket\langle\nu x.\llbracket Q\rrbracket/\alpha\rangle$*, then* $\llbracket P\{\alpha\leftrightarrow\widehat{x}Q\}\rrbracket_\beta = \llbracket P\rrbracket_\beta\langle\nu x.\llbracket Q\rrbracket/\alpha\rangle$.

4. (a) *If* Q *introduces* x*,* $\llbracket P\{\alpha\leftrightarrow\widehat{x}Q\}\rrbracket = \llbracket P\rrbracket\langle\hat{x}(\llbracket Q\rrbracket)/\alpha\rangle$.

   (b) *If* Q *does not introduce* x*, then* $\llbracket P\{\alpha\leftrightarrow\widehat{x}Q\}\rrbracket = \llbracket P\rrbracket\langle\hat{z}([\nu x.\llbracket Q\rrbracket]z)/\alpha\rangle$.

Proof. 1. We consider two cases.

Q **introduces** β**:** Then by Proposition 7.3.2, $\llbracket Q\rrbracket = [\beta]M$ for some $\nu\lambda\mu$-term M with $\beta \notin M$, and $\llbracket Q\rrbracket_\beta = M$. Therefore $\llbracket Q\{P\widehat{\alpha}\leftrightarrow x\}\rrbracket = ([\beta]M)\langle\llbracket P\rrbracket_\alpha/x\rangle = [\beta]M\langle\llbracket P\rrbracket_\alpha/x\rangle$. By Proposition 3.2.6 (4), $Q\{P\widehat{\alpha}\leftrightarrow x\}$ introduces β, and so $\llbracket Q\{P\widehat{\alpha}\leftrightarrow x\}\rrbracket_\beta = M\langle\llbracket P\rrbracket_\alpha/x\rangle$ as required.

Q **does not introduce** β**:** Then $\llbracket Q\rrbracket_\beta = \mu\beta.\llbracket Q\rrbracket$. By Proposition 3.2.6 (4), we conclude that $Q\{P\widehat{\alpha}\leftrightarrow x\}$ does not introduce β. Therefore $\llbracket Q\{P\widehat{\alpha}\leftrightarrow x\}\rrbracket_\beta = \mu\beta.(Q\{P\widehat{\alpha}\leftrightarrow x\}) = \mu\beta.\llbracket Q\rrbracket\langle\llbracket P\rrbracket_\alpha/x\rangle$, by assumption. By definition of substitution then, we conclude $\llbracket Q\{P\widehat{\alpha}\leftrightarrow x\}\rrbracket_\beta = (\mu\beta.\llbracket Q\rrbracket)\langle\llbracket P\rrbracket_\alpha/x\rangle$ as required.

2. By induction on the structure of the term Q. We show here only a representative set of cases.

$Q = \langle x.\beta\rangle$**:**

$$\begin{aligned}
\llbracket Q\{P\widehat{\alpha}\leftrightarrow x\}\rrbracket &= \llbracket P\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle\rrbracket && (\textit{Definition 3.2.4})\\
&= [\beta]\llbracket P\rrbracket_\alpha && (\textit{Definition 7.3.1})\\
&= ([\beta]x)\langle\llbracket P\rrbracket_\alpha/x\rangle && (\textit{substitution})\\
&= \llbracket Q\rrbracket\langle\llbracket P\rrbracket_\alpha/x\rangle && (\textit{Definition 7.3.1})
\end{aligned}$$

$Q = \langle y.\beta\rangle$**:**

$$\begin{aligned}
\llbracket Q\{P\widehat{\alpha}\leftrightarrow x\}\rrbracket &= \llbracket\langle y.\beta\rangle\rrbracket && (\textit{Definition 3.2.4})\\
&= [\beta]y && (\textit{Definition 7.3.1})\\
&= ([\beta]y)\langle\llbracket P\rrbracket_\alpha/x\rangle && (\textit{substitution})\\
&= \llbracket Q\rrbracket\langle\llbracket P\rrbracket_\alpha/x\rangle && (\textit{Definition 7.3.1})
\end{aligned}$$

$Q = Q_1\widehat{\beta}\,[x]\,\widehat{y}Q_2$**:**

$$
\begin{array}{ll}
& \llbracket Q\{P\widehat{\alpha}\leftrightarrow x\}\rrbracket \\
= & \llbracket P\widehat{\alpha} \dagger \widehat{x}((Q_1\{P\widehat{\alpha}\leftrightarrow x\})\widehat{\beta}\,[x]\,\widehat{y}(Q_2\{P\widehat{\alpha}\leftrightarrow x\}))\rrbracket & (\textit{Definition 3.2.4}) \\
= & [\nu y.\llbracket(Q_2\{P\widehat{\alpha}\leftrightarrow x\})\rrbracket]\llbracket(Q_1\{P\widehat{\alpha}\leftrightarrow x\})\rrbracket_\beta & (\textit{Definition 7.3.1}) \\
= & [\nu y.\llbracket Q_2\rrbracket\langle\llbracket P\rrbracket_\alpha/x\rangle]\llbracket(Q_1\{P\widehat{\alpha}\leftrightarrow x\})\rrbracket_\beta & (\textit{by induction}) \\
= & [\nu y.\llbracket Q_2\rrbracket\langle\llbracket P\rrbracket_\alpha/x\rangle]\llbracket Q_1\rrbracket_\beta\langle\llbracket P\rrbracket_\alpha/x\rangle & (\textit{induction and part 1}) \\
= & ([\nu y.\llbracket Q_2\rrbracket]\llbracket Q_1\rrbracket_\beta)\langle\llbracket P\rrbracket_\alpha/x\rangle & (\textit{substitution}) \\
= & \llbracket Q\rrbracket\langle\llbracket P\rrbracket_\alpha/x\rangle & (\textit{Definition 7.3.1})
\end{array}
$$

$Q = \langle x.\beta\rangle\widehat{\beta} \dagger \widehat{y}Q_2$ **:** We distinguish two subcases.

Q_2 **introduces** y**:** Note that since y is bound in Q_2, we may assume $y \notin \mathit{fs}(P)$.

$$
\begin{array}{ll}
& \llbracket Q\{P\widehat{\alpha}\leftrightarrow x\}\rrbracket \\
= & \llbracket P\widehat{\alpha} \dagger \widehat{y}(Q_2\{P\widehat{\alpha}\leftrightarrow x\})\rrbracket & (\textit{Definition 3.2.4}) \\
= & \llbracket(Q_2\{P\widehat{\alpha}\leftrightarrow x\})\rrbracket\langle\llbracket P\rrbracket_\alpha/y\rangle & (\textit{Definition 7.3.1}) \\
= & \llbracket Q_2\rrbracket\langle\llbracket P\rrbracket_\alpha/x\rangle\langle\llbracket P\rrbracket_\alpha/y\rangle & (\textit{by induction}) \\
= & \llbracket Q_2\rrbracket\langle x/y\rangle\langle\llbracket P\rrbracket_\alpha/x\rangle & (\textit{substitution, } y \notin \mathit{fs}(P)) \\
= & \llbracket Q_2\rrbracket\langle\llbracket\langle x.\beta\rangle\rrbracket_\beta/y\rangle\langle\llbracket P\rrbracket_\alpha/x\rangle & (\textit{Definition 7.3.1}) \\
= & \llbracket Q\rrbracket\langle\llbracket P\rrbracket_\alpha/x\rangle & (\textit{Definition 7.3.1})
\end{array}
$$

Q_2 **does not introduce** y**:**

$$
\begin{array}{ll}
& \llbracket Q\{P\widehat{\alpha}\leftrightarrow x\}\rrbracket \\
= & \llbracket P\widehat{\alpha} \dagger \widehat{y}(Q_2\{P\widehat{\alpha}\leftrightarrow x\})\rrbracket & (\textit{Definition 3.2.4}) \\
= & [\nu y.\llbracket(Q_2\{P\widehat{\alpha}\leftrightarrow x\})\rrbracket]\llbracket P\rrbracket_\alpha & (\textit{Definition 7.3.1}) \\
= & [\nu y.\llbracket Q_2\rrbracket\langle\llbracket P\rrbracket_\alpha/x\rangle]\llbracket P\rrbracket_\alpha & (\textit{by induction}) \\
= & ([\nu y.\llbracket Q_2\rrbracket]x)\langle\llbracket P\rrbracket_\alpha/x\rangle & (\textit{substitution, } y \notin \mathit{fs}(P)) \\
= & ([\nu y.\llbracket Q_2\rrbracket]\llbracket\langle x.\beta\rangle\rrbracket_\beta)\langle\llbracket P\rrbracket_\alpha/x\rangle & (\textit{Definition 7.3.1}) \\
= & \llbracket Q\rrbracket\langle\llbracket P\rrbracket_\alpha/x\rangle & (\textit{Definition 7.3.1})
\end{array}
$$

$Q = Q_1\widehat{\beta} \dagger \widehat{y}Q_2$ **and** $Q_1 \neq \langle x.\beta\rangle$**:** We distinguish two subcases.

Q_2 **introduces** y**:** Note that since y is bound in Q_2, we may assume $y \notin fs(P)$.

$$\begin{aligned}
&[\![Q\{P\widehat{\alpha}\leftrightarrow x\}]\!] \\
&= [\![(Q_1\{P\widehat{\alpha}\leftrightarrow x\})\widehat{\beta} \dagger \widehat{y}(Q_2\{P\widehat{\alpha}\leftrightarrow x\})]\!] && (\textit{Definition 3.2.4}) \\
&= [\![(Q_2\{P\widehat{\alpha}\leftrightarrow x\})]\!]\langle [\![Q_1\{P\widehat{\alpha}\leftrightarrow x\}]\!]_\beta / y\rangle && (\textit{Definition 7.3.1}) \\
&= [\![Q_2]\!]\langle [\![P]\!]_\alpha / x\rangle\langle [\![Q_1\{P\widehat{\alpha}\leftrightarrow x\}]\!]_\beta / y\rangle && (\textit{by induction}) \\
&= [\![Q_2]\!]\langle [\![P]\!]_\alpha / x\rangle\langle [\![Q_1]\!]_\beta\langle [\![P]\!]_\alpha / x\rangle / y\rangle && (\textit{induction and part 1}) \\
&= [\![Q_2]\!]\langle [\![Q_1]\!]_\beta / y\rangle\langle [\![P]\!]_\alpha / x\rangle && (\textit{substitution, } y \notin fs(P)) \\
&= [\![Q]\!]\langle [\![P]\!]_\alpha / x\rangle && (\textit{Definition 7.3.1})
\end{aligned}$$

Q_2 **does not introduce** y**:**

$$\begin{aligned}
&[\![Q\{P\widehat{\alpha}\leftrightarrow x\}]\!] \\
&= [\![(Q_1\{P\widehat{\alpha}\leftrightarrow x\})\widehat{\beta} \dagger \widehat{y}(Q_2\{P\widehat{\alpha}\leftrightarrow x\})]\!] && (\textit{Definition 3.2.4}) \\
&= [\nu y.[\![(Q_2\{P\widehat{\alpha}\leftrightarrow x\})]\!]][\![(Q_1\{P\widehat{\alpha}\leftrightarrow x\})]\!]_\beta && (\textit{Definition 7.3.1}) \\
&= [\nu y.[\![Q_2]\!]\langle [\![P]\!]_\alpha / x\rangle][\![(Q_1\{P\widehat{\alpha}\leftrightarrow x\})]\!]_\beta && (\textit{by induction}) \\
&= [\nu y.[\![Q_2]\!]\langle [\![P]\!]_\alpha / x\rangle][\![Q_1]\!]_\beta\langle [\![P]\!]_\alpha / x\rangle && (\textit{induction and part 1}) \\
&= ([\nu y.[\![Q_2]\!]][\![Q_1]\!]_\beta)\langle [\![P]\!]_\alpha / x\rangle && (\textit{substitution}) \\
&= [\![Q]\!]\langle [\![P]\!]_\alpha / x\rangle && (\textit{Definition 7.3.1})
\end{aligned}$$

3. By induction on the structure of the term P, similar to part 1.

4. (a) By induction on the structure of the term P. We show two illustrative cases.

   $P = \langle y.\alpha\rangle$**:**

$$\begin{aligned}
[\![P\{\alpha\leftrightarrow\widehat{x}Q\}]\!] &= [\![\langle y.\alpha\rangle\widehat{\alpha} \dagger \widehat{x}Q]\!] && (\textit{Definition 3.2.4}) \\
&= [\![Q]\!]\langle y/x\rangle && (\textit{Definition 7.3.1}) \\
&= ([\alpha]y)\langle \hat{x}([\![Q]\!])/\alpha\rangle && (\textit{Definition 5.4.5}) \\
&= [\![P]\!]\langle \hat{x}([\![Q]\!])/\alpha\rangle && (\textit{Definition 7.3.1})
\end{aligned}$$

   $P = \widehat{y}P_1\widehat{\beta}\cdot\alpha$**:** By Proposition 3.2.6 (1), we know that $\alpha \notin fs(Q_1\{P\widehat{\alpha}\leftrightarrow x\})$. This fact is used for the step marked $(*)$ below.

$$\begin{aligned}
&[\![P\{\alpha\leftrightarrow\widehat{x}Q\}]\!] \\
&= [\![(\widehat{y}(P_1\{\alpha\leftrightarrow\widehat{x}Q\})\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}Q]\!] && (\textit{Definition 3.2.4}) \\
&= [\![Q]\!]\langle \lambda y.[\![(P_1\{\alpha\leftrightarrow\widehat{x}Q\})]\!]_\beta / x\rangle && (\textit{Definition 7.3.1 and } (*)) \\
&= [\![Q]\!]\langle \lambda y.[\![P_1]\!]_\beta\langle \hat{x}([\![Q]\!])/\alpha\rangle / x\rangle && (\textit{induction and part 3}) \\
&= ([\alpha]\lambda y.[\![P_1]\!]_\beta)\langle \hat{x}([\![Q]\!])/\alpha\rangle && (\textit{Definition 5.4.5}) \\
&= [\![P]\!]\langle \hat{x}([\![Q]\!])/\alpha\rangle && (\textit{Definition 7.3.1})
\end{aligned}$$

   (b) By induction on the structure of the term P, similar to part 2.

□

Since our encoding of $\mathcal{X}^i$-terms into $\nu\lambda\mu$-terms is not compositional, due to the use of substitutions in the encodings, we require the following lemma in order to justify that reduction remains compatible:

Lemma 7.3.8 (Reduction preserved under substitution)**.** *For all $\nu\lambda\mu$-terms M_1, M_2, N and variables x, if $M_1 \to M_2$ then $M_1\langle N/x\rangle \to M_2\langle N/x\rangle$.*

Proof. By induction on the structure of the term M_1. □

We can now prove that, although our encoding is not compositional, reductions which were compatible in the original calculus remain so in the target:

Lemma 7.3.9 (Encoding $\mathcal{X}^i$ to $\nu\lambda\mu$ preserves compatibility)**.** *Let $C[P]$ be an $\mathcal{X}^i$-term with P as a proper sub-term. Let Q be a further $\mathcal{X}^i$-term, and let $C[Q]$ denote the term obtained from $C[P]$ by replacing the subterm P with the term Q (informally, we treat $C[]$ as a context). Then, if $[\![P]\!] \to [\![Q]\!]$ then $[\![C[P]]\!] \to [\![C[Q]]\!]$.*

Proof. By induction on the size of the 'context' $C[\cdot]$. The base case (empty context) is trivial. The inductive cases are mostly immediate, or else follow by Lemma 7.3.8. □

We can finally prove our desired result.

Theorem 7.3.10 (Encoding $\mathcal{X}^i$ to $\nu\lambda\mu$ preserves reductions)**.** *For all $\mathcal{X}^i$-terms P and Q, if $P \to Q$ then $[\![P]\!] \to [\![Q]\!]$.*

Proof. By induction on the length of the reduction sequence $P \to Q$ and on the structure of the term P, using Lemma 7.3.9: we need only consider the case where P is itself a redex, and P reduces to Q in one step. Therefore, we check that for each $\mathcal{X}^i$ reduction rule (Definitions 3.2.3 and 3.2.5), the result holds.

(*cap*)**:** We show $[\![\langle y.\alpha\rangle\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle]\!] \to [\![\langle y.\beta\rangle]\!]$:

$$\begin{aligned}[\![\langle y.\alpha\rangle\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle]\!] &= (([\beta]x))\langle y/x\rangle \\ &= [\beta]y \\ &= [\![\langle y.\beta\rangle]\!]\end{aligned}$$

(*impR*)**:** We show $[\![(\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}\langle x.\gamma\rangle]\!] \to [\![\widehat{y}P\widehat{\beta}\cdot\gamma]\!]$, if $\alpha \notin \mathit{fp}(P)$, which implies that $\alpha \notin \mathit{fv}([\![P]\!])$:

$$\begin{aligned}[\![(\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}\langle x.\gamma\rangle]\!] &= (([\gamma]x))\langle\lambda y.[\![P]\!]_\beta/x\rangle\\ &= [\gamma]\lambda y.[\![P]\!]_\beta\\ &= [\![\widehat{y}P\widehat{\beta}\cdot\gamma]\!]\end{aligned}$$

(*impL*) : We show $[\![\langle y.\alpha\rangle\widehat{\alpha}\dagger\widehat{x}(P\widehat{\beta}\,[x]\,\widehat{z}Q)]\!] \to [\![P\widehat{\beta}\,[y]\,\widehat{z}Q]\!]$, given that $x \notin \mathit{fs}(P,Q)$, i.e. $P\widehat{\beta}\,[x]\,\widehat{z}Q$ introduces x.

$$\begin{aligned}[\![\langle y.\alpha\rangle\widehat{\alpha}\dagger\widehat{x}(P\widehat{\beta}\,[x]\,\widehat{z}Q)]\!] &= (([\nu z.[\![Q]\!]](x\ [\![P]\!]_\beta)))\langle y/x\rangle\\ &= [\nu z.[\![Q]\!]](y\ [\![P]\!]_\beta)\\ &= [\![P\widehat{\beta}\,[y]\,\widehat{z}Q]\!]\end{aligned}$$

(*imp*)**:** Assume that $\alpha \notin \mathit{fp}(P)$ and $x \notin \mathit{fs}(Q,R)$. Then $\alpha \notin \mathit{fv}([\![P]\!])$ and $Q\widehat{\gamma}\,[x]\,\widehat{z}R$ introduces x. We will require as a lemma that $[w][\![P]\!]_\beta \to [\![P]\!]\langle w/\beta\rangle$ (which is easily checked by cases on whether or not P introduces β). We will refer to this Lemma as $(*)$ below. We show that:

$$[\![(\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R)]\!] \to \begin{cases}[\![(Q\widehat{\gamma}\dagger\widehat{y}P)\widehat{\beta}\dagger\widehat{z}R]\!]\\ [\![Q\widehat{\gamma}\dagger\widehat{y}(P\widehat{\beta}\dagger\widehat{z}R)]\!]\end{cases}$$

as follows:

$$\begin{aligned}&[\![(\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R)]\!]\\ &= [\![(Q\widehat{\gamma}\,[x]\,\widehat{z}R)]\!]\langle[\![\widehat{y}P\widehat{\beta}\cdot\alpha]\!]_\alpha/x\rangle\\ &= ([\nu z.[\![R]\!]](x\ [\![Q]\!]_\gamma))\langle[\![\widehat{y}P\widehat{\beta}\cdot\alpha]\!]_\alpha/x\rangle && (\textit{Definition 7.3.1})\\ &= ([\nu z.[\![R]\!]](x\ [\![Q]\!]_\gamma))\langle\lambda y.[\![P]\!]_\beta/x\rangle && (\textit{Definition 7.3.1})\\ &= [\nu z.[\![R]\!]]((\lambda y.[\![P]\!]_\beta)\ [\![Q]\!]_\gamma) && (\textit{substitution})\\ &\to [\nu z.[\![R]\!]]\mu w.[\nu y.[w][\![P]\!]_\beta][\![Q]\!]_\gamma && (\lambda')\end{aligned}$$

From here, we show that the two required reducts can be reached separately. Firstly:

$$\begin{aligned}&[\nu z.[\![R]\!]]\mu w.[\nu y.[w][\![P]\!]_\beta][\![Q]\!]_\gamma\\ &\to [\nu z.[\![R]\!]]\mu\beta.[\nu y.[\![P]\!]][\![Q]\!]_\gamma && (*,\ \alpha\textit{-conversion})\\ &\to [\nu z.[\![R]\!]]\mu\beta.[\![(Q\widehat{\gamma}\dagger\widehat{y}P)]\!] && (\textit{Lemma 7.3.4 (2)})\\ &\to [\![(Q\widehat{\gamma}\dagger\widehat{y}P)\widehat{\beta}\dagger\widehat{z}R]\!] && (\textit{Lemma 7.3.4 (3)})\end{aligned}$$

Secondly:

$$\begin{aligned}
[\nu z.[\![R]\!]]\mu w.[\nu y.[w][\![P]\!]_\beta][\![Q]\!]_\gamma &\to [\nu y.[\nu z.[\![R]\!]][\![P]\!]_\beta][\![Q]\!]_\gamma && (\mu\neg_2)\\
&\to [\nu y.[\![(P\widehat{\beta} \dagger \widehat{z}R)]\!]][\![Q]\!]_\gamma && (\textit{Lemma 7.3.4 (2)})\\
&\to [\![Q\widehat{\gamma} \dagger \widehat{y}[\![(P\widehat{\beta} \dagger \widehat{z}R)]\!]]\!] && (\textit{Lemma 7.3.4 (2)})
\end{aligned}$$

(***not***)**:** We show $[\![(\widehat{y}P \cdot \alpha)\widehat{\alpha} \dagger \widehat{x}(x \cdot Q\widehat{\beta})]\!] \to [\![Q\widehat{\beta} \dagger \widehat{y}P]\!]$, given that $\alpha \notin \mathit{fp}(P)$ and $x \notin \mathit{fs}(Q)$, as follows:

$$\begin{aligned}
[\![(\widehat{y}P \cdot \alpha)\widehat{\alpha} \dagger \widehat{x}(x \cdot Q\widehat{\beta})]\!] &= ([x][\![Q]\!]_\beta)\langle \nu y.[\![P]\!]/x\rangle && (\textit{Definition 7.3.1})\\
&= [\nu y.[\![P]\!]][\![Q]\!]_\beta && (\textit{substitution})\\
&\to [\![Q\widehat{\beta} \dagger \widehat{y}P]\!] && (\textit{Lemma 7.3.4 (2)})
\end{aligned}$$

(*prop*-**R**)**:** We show $[\![P\widehat{\alpha} \dagger \widehat{x}Q]\!] \to [\![Q\{P\widehat{\alpha} \leftarrow\!\!\rightarrow x\}]\!]$, given that Q does not introduce x.

$$\begin{aligned}
[\![P\widehat{\alpha} \dagger \widehat{x}Q]\!] &= [\nu x.[\![Q]\!]][\![P]\!]_\alpha && (\textit{Definition 7.3.1})\\
&\to [\![Q]\!]\langle [\![P]\!]_\alpha/x\rangle && (\nu)\\
&= [\![Q\{P\widehat{\alpha} \leftarrow\!\!\rightarrow x\}]\!] && (\textit{Lemma 7.3.7 (2)})
\end{aligned}$$

(*prop*-**L**)**:** We show $[\![P\widehat{\alpha} \dagger \widehat{x}Q]\!] \to [\![P\{\alpha \leftarrow\!\!\rightarrow \widehat{x}Q\}]\!]$, given that P does not introduce α. We distinguish two cases:

Q introduces x:

$$\begin{aligned}
[\![P\widehat{\alpha} \dagger \widehat{x}Q]\!] &= [\![Q]\!]\langle \mu\alpha.P/x\rangle && (\textit{Definition 7.3.1})\\
&\to [\![P]\!]_{\langle}\hat{x}([\![Q]\!])/\alpha\rangle && (\textit{Corollary 7.3.6})\\
&= [\![P\{\alpha \leftarrow\!\!\rightarrow \widehat{x}Q\}]\!] && (\textit{Lemma 7.3.7 (4)})
\end{aligned}$$

Q does not introduce x:

$$\begin{aligned}
[\![P\widehat{\alpha} \dagger \widehat{x}Q]\!] &= [\nu x.[\![Q]\!]]\mu\alpha.[\![P]\!] && (\textit{Definition 7.3.1})\\
&\to [\![P]\!]\langle \hat{z}([\nu x.[\![Q]\!]]z)/\alpha\rangle && (\mu\neg_2)\\
&= [\![P\{\alpha \leftarrow\!\!\rightarrow \widehat{x}Q\}]\!] && (\textit{Lemma 7.3.7 (4)})
\end{aligned}$$

□

As far as we are aware, this is the first time a full cut-elimination procedure for classical sequent calculus has been encoded into an applicative-style calculus, preserving (the highly non-confluent) reductions. The existence of this encoding relates the problems of strong normalisation of the two calculi, in the sense that if a strong normalisation result

were proven for $\nu\lambda\mu$, our work here would provide an alternative to the work of Urban, for a strong normalisation proof for $\mathcal{X}^i$.

Corollary 7.3.11 (Strong Normalisation of $\nu\lambda\mu$ implies Strong Normalisation of $\mathcal{X}^i$)**.**

1. *If P and Q are $\mathcal{X}^i$-terms such that $P \to Q$ in one step, and $[\![P]\!] = [\![Q]\!]$, then the reduction rule applied in this step must be one of* (cap), $(impR\text{-}rn)$, $(impL\text{-}rn)$, $(not\text{-}left\text{-}rn)$, $(not\text{-}right\text{-}rn)$ *and* (not).

2. *If $P_0, P_1, \ldots$ is any sequence of $\mathcal{X}^i$-terms such that P_0 is typeable, and for all P_j in the sequence with $j > 0$, $P_{j-1} \to P_j$ in one step (in particular $P_{j-1} \neq Pj$), and $[\![P_j]\!] = [\![P_{j+1}]\!]$, then the sequence is necessarily finite.*

3. *If the $\nu\lambda\mu$-calculus satisfies the property that all typeable terms are strongly normalising, then the $\mathcal{X}^i$-calculus does also.*

Proof. 1. By inspection of the proof above, we can see that these are the only rules for which it is possible that the reduction step maps onto identity in the $\nu\lambda\mu$-calculus.

2. Note that by Theorem 3.3.5, each of the P_js remains typeable with the same context which P_0 could be assigned. Since $[\![P_j]\!] = [\![P_{j-1}]\!]$, part 1 enumerates all of the rules which can be applied in the reductions $P_{j-1} \to P_j$. For all of these rules but the (not) rule, the number of cuts in the term decreases. Furthermore, for the (not) rule, the number of cuts stays the same, and the cut removed is replaced by a cut whose type (on the bound connectors) must be of lower degree (i.e., if the redex cut carried type $\neg A$, the new cut carries type A. Therefore, these rules cannot be applied indefinitely to a term; the sequence must eventually terminate.

3. Let P_0 be any typeable $\mathcal{X}^i$-term, and let $P_0, P_1, \ldots$ be any sequence of $\mathcal{X}^i$-terms such that, for all P_j in the sequence with $j > 0$, $P_{j-1} \to P_j$ in one step (in particular $P_{j-1} \neq Pj$). Let $S_0, S_1, \ldots$ be the sequence of maximal subsequences of the sequence $P_0, P_1, \ldots$ satisfying the property that every term in a subsequence S_k maps onto the same $\nu\lambda\mu$-term, under the encoding of Definition 7.3.1. By part 2, each of the subsequences S_k is finite. Let $M_0, M_1, \ldots$ be the corresponding sequence of $\nu\lambda\mu$-terms, i.e., for all P_j in the subsequence S_k, $[\![P_j]\!] = M_k$. Then, by construction, $M_0 = [\![P_0]\!]$, and each pair of successive terms M_k, M_{k+1} in the sequence are distinct from one another. We can repeatedly apply Theorem 7.3.10 to obtain that $M_0 \to M_1 \to \ldots$ is a sequence of $\nu\lambda\mu$-reductions. Furthermore, since P_0 was assumed to be typeable, by Theorem 7.3.3, M_0 is also typeable. If the $\nu\lambda\mu$-calculus satisfies the property that all typeable terms are strongly normalising, then the sequence $M_0 \to M_1 \to \ldots$ must be finite. This means that the

corresponding sequence of subsequences $S_0, S_1, \ldots$ must be finite. Therefore, the original sequence $P_0, P_1, \ldots$ is a finite conjunction of finite sequences, and so is itself finite. Since the sequence was arbitrary, no infinite reduction sequence out of P_0 exists.

□

7.3.1 Additional Reduction Rules

As well as preservation of reductions, it is interesting to note that extra reductions are sometimes possible in the interpreted term, which were not present in the original. For example, an $\mathcal{X}^i$-term of the form $P\widehat{\alpha}\,[x]\,\widehat{y}Q$ in normal form, might (depending on the structures of P and Q) be interpreted as a $\nu\lambda\mu$-term of the form $[\nu y.[\![Q]\!]]\big(x\ (\mu\alpha.[\![P]\!])\big)$, in which there are two redexes: the outer continuation application can be reduced by the rule (ν), while the inner function application can be reduced by the rule $(\mu{\rightarrow_2})$. A similar comment applies to the encoding of terms of the form $x \cdot P\widehat{\alpha}$; the resulting term $[x][\![P]\!]_\alpha$ has an extra $(\mu{\neg_2})$ redex in the case where P does not introduce α. is not of the form $[\alpha]M$ with $\alpha \notin M$. These extra reductions in the encoded term suggest the following extra reduction rules could be added to $\mathcal{X}^i$:

Definition 7.3.12 (Possible extra reduction rules for $\mathcal{X}^i$)**.**

$$
\begin{array}{lll}
(\textit{imp-extra-1})\ P\widehat{\alpha}\,[x]\,\widehat{y}Q \rightarrow P\{\alpha{\leftrightarrow}\widehat{z}(\langle z.\beta\rangle\widehat{\beta}\,[x]\,\widehat{y}Q)\} & (\textit{if } P \textit{ does not introduce } \alpha) \\
(\textit{imp-extra-2})\ P\widehat{\alpha}\,[x]\,\widehat{y}Q \rightarrow Q\{(P\widehat{\alpha}\,[x]\,\widehat{z}\langle z.\beta\rangle)\widehat{\beta}{\leftrightarrow}y\} & (\textit{if } Q \textit{ does not introduce } y) \\
(\textit{not-extra}) \qquad x \cdot P\widehat{\alpha} \rightarrow P\{\alpha{\leftrightarrow}\widehat{y}(x \cdot \langle y.\beta\rangle\widehat{\beta})\} & (\textit{if } P \textit{ does not introduce } \alpha)
\end{array}
$$

The most obvious objection to the possible inclusion of these reduction rules is that they break the 'cut=redex' paradigm usual for sequent calculus (and present in the definition of the $\mathcal{X}^i$-calculus). However, it is still interesting to consider their implications: they do make sense in terms of the types, and make a kind of intuitive sense also. For example, if there are many occurrences of y in Q, then the second rule above allows these to be 'sought out' before any cut with the ys in Q is actually built. So this appears to be a kind of 'look-ahead' rule; it anticipates the behaviour that would eventually be possible if the *impL* term were reduced in a cut by the rule $(\textit{imp})$.

However, it turns out that the rules formulated above are somewhat naïve: if the two rules $(\textit{imp-extra-1})$ and $(\textit{imp-extra-2})$ were both added as reduction rules, then strong normalisation of typeable terms would immediately be violated, in a similar way to that shown in Definition 3.2.7. In particular, a term of the form $P\widehat{\alpha}\,[x]\,\widehat{y}(\langle y.\beta\rangle\widehat{\beta}\,[z]\,\widehat{w}R)$ could

be constructed which runs by the first rule to $(P\widehat{\alpha}\ [x]\ \widehat{y}\langle y.\beta\rangle)\widehat{\beta}\ [z]\ \widehat{w}R$, which runs by the second rule back to the original, creating a loop. This does not imply that the $\nu\lambda\mu$-calculus itself has such looping reductions: instead, the rules above do not accurately reflect the reductions of $\nu\lambda\mu$. If we try to simulate the same looping behaviour using the encoded versions of the terms, we can see the inconsistency:

$$\begin{aligned}
\llbracket P\widehat{\alpha}\ [x]\ \widehat{y}(\langle y.\beta\rangle\widehat{\beta}\ [z]\ \widehat{w}R)\rrbracket &= [\nu y.\llbracket(\langle y.\beta\rangle\widehat{\beta}\ [z]\ \widehat{w}R)\rrbracket]\big(x\ \llbracket P\rrbracket_\alpha\big) \\
&= [\nu y.[\nu w.\llbracket R\rrbracket](z\ y)]\big(x\ \llbracket P\rrbracket_\alpha\big) \\
&\to [\nu w.\llbracket R\rrbracket]\big(z\ \big(x\ \llbracket P\rrbracket_\alpha\big)\big) \qquad (\nu) \\
&\leftarrow [\nu w.\llbracket R\rrbracket]\big(z\ (\mu\beta.[\nu y.[\beta]y]\big(x\ \llbracket P\rrbracket_\alpha\big)\big)\big) \\
&= \llbracket(P\widehat{\alpha}\ [x]\ \widehat{y}\langle y.\beta\rangle)\widehat{\beta}\ [z]\ \widehat{w}R\rrbracket
\end{aligned}$$

In order to obtain the 'loop', we have to make some expansions in the $\nu\lambda\mu$-reduction. Therefore, although the possible extra rules for $\mathcal{X}^i$ are too problematic to be included, the reductions present in $\nu\lambda\mu$ do not exhibit the same problems. We have not identified a useful restriction of the three 'extra' rules above, which does not create such loops. The main reason for considering the addition of these extra rules, is that they (or some better-behaved variants) actually appear to be necessary to make an encoding back from $\nu\lambda\mu$ to $\mathcal{X}^i$ viable. These, and other issues, are discussed further in the next section.

7.4 Some Thoughts on Encoding $\nu\lambda\mu$ into $\mathcal{X}^i$

We have not been able to find an analogous 'inverse encoding' to that of the previous section. This appears to be because the $\nu\lambda\mu$-calculus is rather *too* expressive; there are terms which do not have 'obvious' analogues in the $\mathcal{X}^i$-calculus, and reduction rules which do not easily correspond to cut elimination steps. This seems somewhat surprising, since the logical origins of both calculi are fairly clear. We discuss some of the issues encountered here.

7.4.1 Making $\bot$ an Explicit Connective

Since our $\nu\lambda\mu$-syntax does not provide names for outputs, any encoding of $\nu\lambda\mu$ into $\mathcal{X}^i$ should take a plug as a parameter, which, except in the case of a 'silent' term (one which is necessarily of type $\bot$), provides an explicit name for the output of the term. For example, we might interpret the variable x with respect to the plug α (written $\llbracket x\rrbracket_\alpha$) as the $\mathcal{X}^i$-term $\langle x.\alpha\rangle$, in which the output name is explicit. In the case of a 'silent' term, this name can

still be provided, but should be ignored by the encoding. For example, we might interpret $[x]y$ with respect to the plug α as $x \cdot \langle y.\beta\rangle\widehat{\beta}$, which does not mention α (and indeed, has no free outputs).

In some cases, we wish to *enforce* that a term be 'silent'. For example, in $\nu\lambda\mu$ we know that the body of a μ- or ν-abstraction must (if it is typeable) necessarily be of type $\bot$. This lack of an output type is essential for the soundness of the μ-reduction rules. We wish this fact to be reflected in an encoding, also. For example, it might seem tempting to define $[\![\nu x.M]\!]_\alpha = \widehat{x}[\![M]\!]_\beta \cdot \alpha$. This is indeed the right idea, so long as M is 'silent' (i.e. β does not occur in the result). But consider the $\nu\lambda\mu$-term $\nu x.x$. When we encode into $\mathcal{X}^i$, using the rule above, we introduce a free occurrence of β from the body of the ν-bound term: $[\![\nu x.x]\!]_\alpha = \widehat{x}\langle x.\beta\rangle \cdot \alpha$. This fails to respect our idea that the inner-subterm should be 'silent'. Furthermore, the choice of β as a free name is arbitrary (α was mentioned in the original encoding, but not β), and so we have a rather-strange non-determinism in the resulting encoding.

One way to get around this problem, is to explicitly represent 'silent variables' in a different manner. An easy way to manage this is to add $\bot$ as an explicit connective in the logic underlying $\mathcal{X}^i$. This implies the addition of $\bot$ into the language of types. To manage this addition in the inference rules, it is sufficient to add a left-introduction rule for $\bot$ (there is no corresponding right-introduction rule). The inference rule is the following:

$$\frac{}{\Gamma, x : \bot \vdash \Delta}\ (\bot\mathcal{L})$$

Since this rule introduces a statement on the left of the sequent (but nothing else), and has no premises, it should be inhabited by a syntax construct which introduces a free socket (but not plug), and has no subterms. This construct should resemble a capsule whose output is 'silent', and we use the notation $\langle x.\bullet\rangle$ for it. Therefore, we could add terms of this form to the syntax of $\mathcal{X}^i$, and add the following rule to the type system:

$$\frac{}{\langle x.\bullet\rangle :\cdot \Gamma, x : \bot \vdash \Delta}\ (\bot\mathcal{L})$$

Naturally, a term of the form $\langle x.\bullet\rangle$ introduces the socket x and no other connector. The only additional reduction rule associated with this syntax construct is a rule for renaming with capsules:

$$(\textit{bot-rn}) :\ \langle x.\alpha\rangle\widehat{\alpha} \dagger \widehat{y}\langle y.\bullet\rangle \rightarrow \langle x.\bullet\rangle$$

There is no rule which explicitly removes a $\langle x.\bullet\rangle$ construct from a term (in contrast to the other syntax constructs, who all have logical rules to unravel them). This means that the only way a $\langle x.\bullet\rangle$ construct can be removed during reduction is if it is bound in a cut such as $P\widehat{\alpha} \dagger \widehat{x}\langle x.\bullet\rangle$ in which α does not occur in P.

With the new syntax construct present, we can now define an operation to 'silence' an output in an $\mathcal{X}^i$-term:

Definition 7.4.1 (Silencing an output)**.** *We define the mapping $(P)_{\alpha\mapsto\perp}$ recursively on the structure of P as follows:*

$$\begin{aligned}
(\langle x.\bullet\rangle)_{\alpha\mapsto\perp} &= \langle x.\bullet\rangle \\
(\langle x.\alpha\rangle)_{\alpha\mapsto\perp} &= \langle x.\bullet\rangle \\
(\langle x.\beta\rangle)_{\alpha\mapsto\perp} &= \langle x.\beta\rangle && \beta \neq \alpha \\
(\widehat{x}Q\widehat{\beta}\cdot\alpha)_{\alpha\mapsto\perp} &= (\widehat{x}((Q)_{\alpha\mapsto\perp})\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{y}\langle y.\bullet\rangle \\
(\widehat{x}Q\widehat{\beta}\cdot\gamma)_{\alpha\mapsto\perp} &= \widehat{x}((Q)_{\alpha\mapsto\perp})\widehat{\beta}\cdot\gamma, && \gamma \neq \alpha \\
(Q\widehat{\beta}\,[y]\,\widehat{x}R)_{\alpha\mapsto\perp} &= ((Q)_{\alpha\mapsto\perp})\widehat{\beta}\,[y]\,\widehat{x}((R)_{\alpha\mapsto\perp}) \\
(\widehat{x}Q\cdot\alpha)_{\alpha\mapsto\perp} &= (\widehat{x}((Q)_{\alpha\mapsto\perp})\cdot\alpha)\widehat{\alpha} \dagger \widehat{y}\langle y.\bullet\rangle \\
(\widehat{x}Q\cdot\gamma)_{\alpha\mapsto\perp} &= \widehat{x}((Q)_{\alpha\mapsto\perp})\cdot\gamma, && \gamma \neq \alpha \\
(y\cdot Q\widehat{\beta})_{\alpha\mapsto\perp} &= y\cdot((Q)_{\alpha\mapsto\perp})\widehat{\beta} \\
(Q\widehat{\beta}\dagger\widehat{x}R)_{\alpha\mapsto\perp} &= ((Q)_{\alpha\mapsto\perp})\widehat{\beta}\dagger\widehat{x}((R)_{\alpha\mapsto\perp})
\end{aligned}$$

It is straightforward to show that $\alpha \notin \mathit{fp}((P)_{\alpha\mapsto\perp})$, for any term P. We can now resolve the difficulty we previously described concerning the encoding of terms which we wish to be 'silent'. Rather than the arbitrary occurrence of a new free plug in the result, we 'silence' this plug (therefore the exact choice of the name is irrelevant). Returning to our previous example, we now define $[\![\nu x.M]\!]_\alpha = \widehat{x}([\![M]\!]_\beta)_{\beta\mapsto\perp}\cdot\alpha$. For example, this would mean that $[\![\nu x.x]\!]_\alpha = \widehat{x}\langle x.\bullet\rangle\cdot\alpha$.

Unfortunately, although this approach seems promising, it is not sufficient. The difficulty we come across when encoding the full $\nu\lambda\mu$ calculus is illustrated by case of the (ν) reduction rule. If we consider a simple case, such as $[\nu x.x]y \to y$ (which is only typeable by making y a variable of type $\perp$), and encode the two terms via a plug α, we can see the problem precisely. The encoding of the reduct via α will (according to the discussion above), be $[\![[\nu x.x]y]\!]_\alpha = [\![y]\!]_\beta\widehat{\beta} \dagger \widehat{x}[\![x]\!]_\perp = \langle y.\beta\rangle\widehat{\beta} \dagger \widehat{x}\langle x.\bullet\rangle$. This $\mathcal{X}^i$-term reduces to $\langle y.\bullet\rangle$. But, the encoding of the original redex, y, via the plug α, gives $\langle y.\alpha\rangle$ instead. Therefore, we have a mismatch. The problem is that, although we were able to see that the bound variable x in the original term must be 'silent', this information is lost in the reduction, and the variable y now appears just as a normal variable (which could be typeable with any type, not just $\perp$). The encoding reflects this loss of information too,

and so we obtain an inconsistent result.

We have not found a way to avoid this problem for $\nu\lambda\mu$ in general. However, one could attempt to give an encoding which works for a subset of $\nu\lambda\mu$. In particular, if we wish to avoid the problem described above, we need to be sure that the terms which are 'silenced' by the encoding remain 'silent' after reduction. One way in which this can be achieved, is to restrict the bodies of ν- and μ-abstraction to always be continuation applications (terms of the form $[M]N$). This would not be very pleasing regarding the original ambitions of $\nu\lambda\mu$ to represent the full logic, but still provides a very rich syntax, which is closed under reductions. Furthermore, the image of $\mathcal{X}^i$, under the encoding presented in the previous section, falls within this syntax, suggesting that it might be an interesting restriction to study.

7.4.2 Inputs to Outputs

Since we relate μ-reductions with left-propagation of cuts, we need to design any encoding so that the encoding of a μ-bound term results in an $\mathcal{X}^i$-term with occurrences of *outputs* rather than inputs, corresponding to the occurrences of its bound variable. This is also related to our desire to present an 'inverse' to the encoding in the previous section: there, all outputs α in the original $\mathcal{X}^i$-term and mapped onto occurrences of a corresponding variable α. In order for the inverse encoding to reach the original term, we must somehow map these variable occurrences back on to outputs. In particular, since outputs α of type A map on to variables α of type $\neg A$, we now require an operation to replace occurrences of inputs of type $\neg A$ with outputs of type A. We observe that this can be achieved as a 'second phase': firstly an $\nu\lambda\mu$-term is encoded into $\mathcal{X}^i$, and then, where appropriate, a transformation is applied on the resulting term to replace inputs with outputs. In the $\mathcal{X}^i$-setting, there is an easy way in which this can be achieved. If P is an $\mathcal{X}^i$-term in which x occurs with type $\neg A$, and in which β does not occur, then $P\{(\widehat{y}\langle y.\beta\rangle \cdot \alpha)\widehat{\alpha}{\leftrightarrow}x\}$ is an $\mathcal{X}^i$-term in which β occurs with type A.

7.5 Shallow Polymorphism for $\nu\lambda\mu$

A different application of our understanding of the relationship between the $\mathcal{X}^i$ and $\nu\lambda\mu$-calculi, is that we can consider how to adapt the results of Chapter 4 to the $\nu\lambda\mu$-calculus. In particular, Theorem 7.3.3 gives us a strong relationship between the typings possible in the simple type systems for both calculi. We can use this intuition as the basis of an extension of the $\nu\lambda\mu$-calculus to shallow polymorphism. We would anticipate subject reduction

to be just as problematic as in the case of the $\mathcal{X}^i$-calculus, but since we have already found an elegant solution to those problems, the adaptation is relatively straightforward.

One interesting observation which becomes quickly clear is that an analogous type system for $\nu\lambda\mu$ cannot deal with strictly shallow types (i.e., it is no longer sufficient to consider only types with quantifiers on the outside). However, the extension is not very great, and rather natural, as the following example should illustrate.

Consider the self-application of the identity function. This is represented by the $\mathcal{X}^i$-term $(\widehat{x}\langle x.\alpha\rangle\widehat{\alpha}\cdot\beta)\widehat{\beta} \dagger \widehat{y}(\langle y.\gamma\rangle\widehat{\gamma}\ [y]\ \widehat{z}\langle z.\delta\rangle)$, which is typeable according to Definition 4.3.6 in the following way:

$$\dfrac{\dfrac{\dfrac{}{\langle x.\alpha\rangle\ \vdots\cdot\ x:\varphi'\vdash_{\mathrm{SP}}\alpha:\varphi'}\,(ax)}{\widehat{x}\langle x.\alpha\rangle\widehat{\alpha}\cdot\beta\ \vdots\cdot\ \emptyset\vdash_{\mathrm{SP}}\beta:\forall X.(X\to X)}\,(\to\mathcal{R}) \quad \dfrac{\dfrac{}{\langle y.\gamma\rangle\ \vdots\cdot\ y:\forall X.(X\to X)\vdash_{\mathrm{SP}}\gamma:\varphi\to\varphi}\,(ax) \quad \dfrac{}{\langle z.\delta\rangle\ \vdots\cdot\ z:\varphi\to\varphi\vdash_{\mathrm{SP}}\delta:\varphi\to\varphi}\,(ax)}{\langle y.\gamma\rangle\widehat{\gamma}\ [y]\ \widehat{z}\langle z.\delta\rangle\ \vdots\cdot\ y:\forall X.(X\to X)\vdash_{\mathrm{SP}}\delta:\varphi\to\varphi}\,(\to\mathcal{L})}{(\widehat{x}\langle x.\alpha\rangle\widehat{\alpha}\cdot\beta)\widehat{\beta}\dagger\widehat{y}(\langle y.\gamma\rangle\widehat{\gamma}\ [y]\ \widehat{z}\langle z.\delta\rangle)\ \vdots\cdot\ \emptyset\vdash_{\mathrm{SP}}\delta:\varphi\to\varphi}\,(cut)$$

Now, if we encode this term into the $\nu\lambda\mu$-calculus, by applying Definition 7.3.1, we hope the resulting term can be typeable in a suitably-defined analogous polymorphic type system. Applying the definition, we have

$$\llbracket(\widehat{x}\langle x.\alpha\rangle\widehat{\alpha}\cdot\beta)\widehat{\beta} \dagger \widehat{y}(\langle y.\gamma\rangle\widehat{\gamma}\ [y]\ \widehat{z}\langle z.\delta\rangle)\rrbracket_\delta = \mu\delta.[\nu y.[\nu z.[\delta]z](y\ y)]\lambda x.x$$

If we reduce the inner (ν) redex in the resulting term, we reach something looking reasonably close to the original: $\mu\delta.[\nu y.[\delta](y\ y)]\lambda x.x$. Note that, as is typical of our encoding, the cut in the original term has become a continuation application in the $\nu\lambda\mu$-term. However, in the original term the cut carried the shallow polymorphic type $\forall X.(X\to X)$. If we want to use this type in the resulting term, the subterm $\lambda x.x$ may reasonably be typed with it, but the subterm $\nu y.[\delta](y\ y)$ needs instead the type $\neg(\forall X.(X\to X))$. This is not a shallow-polymorphic type. Similarly, if $\lambda x.x$ could be assigned the type $\forall X.(X\to X)$, intuitively we would expect that a term $\mu w.[w]\lambda x.x$ could do as well. But, in constructing this derivation, we will need an occurrence of the negated shallow polymorphic type $\neg(\forall X.(X\to X))$. Thismotivates an extension to the language of types and type-schemes, allowing possibly-negated type schemes $\overline{\overline{A}}$:

$$\begin{aligned} A, B &::= \varphi \mid X \mid (A \to B) \\ \overline{A} &::= \forall X_1.\forall X_2.\ldots\forall X_n.A \ \ (n \geq 0) \\ \overline{\overline{A}} &::= \overline{A} \ \mid\ \neg\overline{A} \end{aligned}$$

Given this extension, we then consider the following type-derivation to be the counterpart of that shown for the original $\mathcal{X}^i$ term:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\delta : \neg(\varphi{\rightarrow}\varphi) \vdash \delta : \neg(\varphi{\rightarrow}\varphi)}\,(ax) \quad \dfrac{}{y : \forall X.(X{\rightarrow}X) \vdash y : (\varphi{\rightarrow}\varphi){\rightarrow}(\varphi{\rightarrow}\varphi)}\,(ax) \quad \dfrac{}{y : \forall X.(X{\rightarrow}X) \vdash y : \varphi{\rightarrow}\varphi}\,(ax)}{y : \forall X.(X{\rightarrow}X) \vdash y\ y : \varphi{\rightarrow}\varphi}\,(\rightarrow\mathcal{E})}{\delta : \neg(\varphi{\rightarrow}\varphi), y : \forall X.(X{\rightarrow}X) \vdash [\delta](y\ y) : \bot}\,(\neg\mathcal{E}) \quad \dfrac{}{x : \varphi' \vdash x : \varphi'}\,(ax)}{\delta : \neg(\varphi{\rightarrow}\varphi) \vdash \nu y.[\delta](y\ y) : \neg(\forall X.(X{\rightarrow}X))}\,(\neg\mathcal{I}) \quad \dfrac{}{\vdash \lambda x.x : \forall X.(X{\rightarrow}X)}\,(\rightarrow\mathcal{I})}{\delta : \neg(\varphi{\rightarrow}\varphi) \vdash [\nu y.[\delta](y\ y)]\lambda x.x : \bot}\,(\neg\mathcal{E})}{\emptyset \vdash \mu\delta.[\nu y.[\delta](y\ y)]\lambda x.x : \varphi{\rightarrow}\varphi}\,(PC)$$

We can formally define the analogous type system for $\nu\lambda\mu$ as follows:

Definition 7.5.1 (Shallow-polymorphic type assignment for $\nu\lambda\mu$-calculus)**.**

$$\frac{}{\Gamma, x : \overline{\overline{A}} \vdash x : \overline{\overline{B}}}\,(Ax)^1 \qquad \frac{\Gamma, x : \neg\overline{A} \vdash M : \bot}{\Gamma \vdash \mu x.M : \overline{A}}\,(\boldsymbol{PC})$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : \overline{C}}\,(\rightarrow\mathcal{I})^2 \qquad \frac{\Gamma \vdash M : A{\rightarrow}B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\,(\rightarrow\mathcal{E})$$

$$\frac{\Gamma, x : \overline{A} \vdash M : \bot}{\Gamma \vdash \nu x.M : \overline{B}}\,(\neg\mathcal{I})^3 \qquad \frac{\Gamma \vdash M : \neg\overline{A} \quad \Gamma \vdash N : \overline{A}}{\Gamma \vdash [M]N : \bot}\,(\neg\mathcal{E})$$

[1] $\overline{\overline{A}} \succeq \overline{\overline{B}}$. [2] $(A{\rightarrow}B) \lhd_\Gamma \overline{C}$. [3] *either* $\overline{B} = \neg\overline{A}$ *or* $\overline{A} = A$ *and* $(\neg A) \lhd_\Gamma \overline{B}$.

This type system follows analogous restrictions on polymorphic generalisation to that presented for the $\mathcal{X}^i$-calculus (Definition 4.3.6). Quantifiers can be added to the types of λ-abstractions and ν-abstractions, so long as it is sound to do so (the predicate $\overline{A} \lhd_\Gamma \overline{B}$ is the obvious adaptation of that of Definition 4.3.4), and, in the case of ν-abstraction, so long as we do not go outside of our type language. This extra restriction is needed because we *allow* the type of the ν-bound variable to potentially be a type scheme (in contrast with λ-binding), in order to model the substitution of polymorphic values into contexts.

The fact that μ-bound terms may also carry polymorphic types, means that (just as in the case of Chapter 4), terms of polymorphic type are not restricted to values. Indeed, the term $\mu x.((w\ ([x]\lambda y.y))\ ([x]\lambda z.z))$ (in which the variable w is just 'dummy' structure for the purpose of the example), contains two copies of the polymorphic identity function, and can be typed as such according to the typing rules above (we leave the derivation to the keen reader) as $w : \bot{\rightarrow}\bot{\rightarrow}\bot \vdash \mu x.((w\ ([x]\lambda y.y))\ ([x]\lambda z.z)) : \forall X.(X{\rightarrow}X)$.

We have not proved properties of this proposed type system for $\nu\lambda\mu$, but it seems that subject reduction and principal contexts can be dealt with analogously to the work of Chapter 4. If so, we have a fairly general and sound decidable polymorphic type system for a term calculus based on classical natural deduction, which we believe to be a new result. Even when compared with related calculi, such as ML with call/cc, we believe our system to be (slightly) more permissive than, for example, the proposal of Wright [105]. We believe that the particular type system described above would not have been easy to arrive at by working directly in the natural deduction paradigm (in particular, the conditions on when negated type schemes can and cannot occur, are subtle), but the analogous system arose naturally through our work in the sequent calculus paradigm, in which the unsoundness of the naïve system can be understood clearly.

7.6 Confluent Restrictions of $\nu\lambda\mu$

One of the results we hoped to achieve by defining encodings between our two calculi was to provide a clear definition and explanation of the call-by-name and call-by-value restrictions of our $\nu\lambda\mu$-calculus. As was discussed in Chapter 3, there is a simple definition of these confluent restrictions in the context of the sequent calculus, which is obtained by favouring either left or right propagation of cuts when both are possible (c.f. Definition 3.5.1). Our original plan was that, armed with a suitable encoding of $\nu\lambda\mu$ into $\mathcal{X}^i$, we could define the call-by-name and call-by-value restrictions of $\nu\lambda\mu$ by encoding redexes into the sequent calculus, and examining what the restrictions there naturally implied for the original term. This approach is not possible, since we have not obtained such an encoding, however we have at least reached a partial understanding of how computations may be related between the two disciplines, through our encoding in the other direction. In particular, we have identified that left-propagation of cuts in $\mathcal{X}^i$, and mu-reductions in $\nu\lambda\mu$ are closely related (while, less surprisingly, right-propagation and term substitution also approximately correspond).

We consider here a more intensional definition of the call-by-name and call-by-value disciplines, and show that it can be generalised to provide an understanding of our particular problem, but also applied to other calculi. In doing so, we observe that the existing definitions of call-by-name $\lambda\mu$-calculus are restricted beyond the 'natural' definitions, and argue that, in fact, call-by-value restrictions of the $\lambda\mu$ and $\nu\lambda\mu$-calculi are *not* naturally confluent restrictions.

The most well-known example of these subsystems is the call-by-value λ-calculus. As is well-known, this is obtained from the full λ-calculus by restricting the β-rule to only

apply when the function argument is a *value*, where values are either λ-abstractions or variables. In other cases, reduction is blocked, for example $(\lambda x.x)\ (y\ z)$ is a normal form in CBV λ-calculus. Initially, it might seem somewhat surprising that variables are considered values, since, in principle, this means that the set of values is not closed under substitution. Therefore, it might be thought possible that, although the example above is in normal form, the term $(\lambda x.x)\ w$ runs to w, and, if $y\ z$ is then substituted for w, then the original 'block' to reduction seems to be evaded. The reason that this kind of flaunting of the intentions of CBV does not occur is that, because of exactly the same restriction, the only *substitutions* which are ever generated by CBV reduction replace variables with *values*. Therefore, it is in fact the case that the syntactic category of values is closed under CBV substitutions.

We consider then, that one could define CBV λ-calculus in another way. Instead of beginning with the value restriction, we could obtain the same calculus by requiring that all *substitutions* generated by reduction must always be of *values* for variables. This restriction would then implicitly define the same restriction of the β rule as is well-known. Why then, do we consider this to be advantageous? The point is that the very same definition serves to implicitly define a notion of call-by-value for other calculi. For example, in the context of the $\overline{\lambda}\mu\tilde{\mu}$-calculus (c.f., Section 5.5.4), there is a term-for-variable substitution similar to that in the λ-calculus. If one imposes the restriction that the only substitutions of this kind which may be generated are those replacing variables with *values*, then one immediately obtains the restriction that the $(\tilde{\mu})$ reduction rule may not be applied when the left-hand term of the command is not a value (which is equivalent to it being a μ-bound term, in this calculus). This is exactly the restriction imposed by Curien and Herbelin to define the CBV version of $\overline{\lambda}\mu\tilde{\mu}$. Dually, in order to obtain CBN $\overline{\lambda}\mu\tilde{\mu}$, one can insist that substitutions of contexts for context-variables are only generated when the context is an "applicative context" [21], which is essentially a dual notion to that of terms being values. This restriction on the second class of substitutions, naturally implies the definitions of $\overline{\lambda}\mu\tilde{\mu}$. Note that, for each of the two subsystems, it is only necessary to impose a restriction on *one* of the two classes of substitution, leaving the other unchanged.

Now we consider this idea for the $\nu\lambda\mu$-calculus. We focus on the case of call-by-value, since it is simpler to reason about in our setting. We argue that to define CBV $\nu\lambda\mu$, we need only insist that term substitutions may never be generated unless the term replacing the variable is a value, where values are λ-abstractions, ν-abstractions and variables. This implies that the (ν) reduction rule (which is the only rule which generates term-for-variable substitutions) should be restricted to be only applicable when the right-hand subterm (argument is a value), i.e.:

$$(\nu_{CBV}) : [\nu x.M]V \to M\langle V/x\rangle$$

According to our pattern, no other restrictions should be necessary; the other reduction rules largely generate semi-structural substitutions, which we would only expect to restrict to define call-by-name. In particular, note that we do not appear to need to restrict the (λ') rule, since this rule does not generate *any* kind of substitution. This can also be compared to the situation in $\overline{\lambda}\mu\tilde{\mu}$, where the $(\to')$ rule is not restricted in either subcalculus [21]. However, this approach does not suggest that any of the μ-reductions of the calculus need naturally be restricted in call-by-value. For example, the two rules (c.f. Definition 5.4.6):

$$(\mu\to_1)\ (\mu x.M)\ N \to \mu y.M\langle\hat{z}([y](z\ N))/x\rangle$$
$$(\mu\to_2)\ N\ (\mu x.M) \to \mu y.M\langle\hat{z}([y](N\ z))/x\rangle$$

seem both to be compatible (according to the reasoning above) with the idea of call-by-value reduction. This suggests that the 'natural' notion of call-by-value reduction in this calculus is still non-confluent. This would explain the observation by Rocheteau [81] that some authors have defined CBV $\lambda\mu$-calculus with one rule present, and some with the other; essentially the choice between the two could be viewed as an arbitrary restriction imposed to guarantee confluence of "call-by-value" reduction. We believe it is interesting to note that a reduction system could appear to be naturally call-by-value and still naturally non-confluent, in the setting of classical logic.

Since we have claimed that μ-reductions in $\nu\lambda\mu$ correspond with left-propagation reductions in the setting of classical sequent calculus, it is natural to ask whether the discussion above implies that the notion of call-by-value reduction which we consider for the $\mathcal{X}^i$-calculus (Definition 3.5.1) is also non-confluent. The reason this is not an obvious consequence is tied up with the problems we have with encoding $\nu\lambda\mu$ into $\mathcal{X}^i$; the second of the two rules above cannot easily be encoded, and in fact corresponds informally with one of the possible extra reduction rules discussed in Definition 7.3.12. Therefore, there is not an obvious counter-example to the confluence of call-by-name reduction in $\mathcal{X}^i$, unless these extra rules were to be added.

7.7 Summary

We have presented an encoding of a fully-general cut-elimination procedure for classical sequent calculus, into a Gentzen-style natural deduction paradigm. Concretely, we have encoded the $\mathcal{X}^i$-calculus into the $\nu\lambda\mu$-calculus, in such a way that reductions and typings

are preserved. We have investigated the possibility of an encoding in the other direction, but have not, thus far, been able to obtain one. However, the reduction rules which cannot easily be simulated lead naturally to the consideration of addition of extra reduction rules in the sequent calculus paradigm, which seems an interesting area of future work.

It is also interesting to consider whether such encodings could be extended to the case of 'explicit' substitution operations. For example, the $\mathcal{X}$-calculus [98] is based on the 'localised version' of Christian Urban's cut elimination [92], in which propagation of cuts is modelled step-by-step by the reduction rules, in much the same manner as explicit substitutions in the λx-calculus [18]. A natural question to ask is whether the encoding we have presented could be easily extended to an encoding of the $\mathcal{X}$-calculus. This would (at least) require the addition of explicit substitution operators into the term syntax of the $\nu\lambda\mu$-calculus, but this should be achievable without much effort. Previous work already shows that the incorporation of explicit substitutions into the $\lambda\mu$-calculus [8], and the related $\lambda\Delta$-calculus [14], is possible without breaking the good properties of the calculi. However, we observe that the direct preservation of cut-elimination reductions becomes problematic in this case. This is essentially because the exact orders of propagating 'substitutions' in the two paradigms do not always match up. Therefore, it is not easy to achieve the result that every reduction step in the source calculus is modelled in the target calculus: we can only show that once the substitutions have been fully evaluated, then the reductions match up. These technical difficulties are evaded in our work, since we abstract away from the step-by-step propagation of substitution operations.

Chapter 8

Conclusions

Throughout this thesis, we have been concerned with the investigation of computational content for the classical logics presented by Gentzen. We have, with the notable exception of Kleene's permutation-free presentation of the sequent calculus, adhered strongly to the viewpoint that it is interesting to consider this subject with respect to the original presentations of the logics, and in full generality. In particular, the natural non-confluence which pervades this work is regarded as an essential ingredient of general calculi based on classical logics, and we allow it to be fully expressed in our work. Aesthetic considerations aside, this attitude can be justified by the argument that, in order to understand exactly what the natural computational content in these logics might be, we should work without restricting the notions of reductions from their most-general forms. We believe it is valuable to consider what these notions of reduction might mean in general, before (if necessary) restricting them to obtain a suitable fragment for further study.

We began by outlining a brief history of the work which has chiefly contributed to this point of view. While the seminal work of Griffin [43] and the subsequent work by Parigot [66, 68] and many other authors was essential to the development of this research field, we believe it is the work of Christian Urban [92] which most-conclusively argues that a meaningful and expressive computational content for classical logic can (and should) be extracted with a non-confluent set of reductions. Indeed, rather than arguing to restrict to a unique normal form, he aims to extract *as many* normal forms as possible from a given proof. His set of reductions, while not entirely complete (in the sense that some "potential" normal forms are not reachable, as he discusses in the conclusion of his thesis), seem to give the most general notion of cut elimination for classical sequent calculus which has been proved to be strongly normalising.

We used the work of Urban as the basis of our chosen term calculus based on classical sequent calculus, and presented an untyped variation of his work, $\mathcal{X}^i$, incorporating the

infix notation introduced by van Bakel et. al. [98]. The resulting calculus is automatically known to satisfy witness reduction and strong normalisation, and because of its strong ties to the logic and general notion of reduction, we use it as the basis of our work on classical sequent calculus. We showed other standard properties, including a notion of principal typing for the simple type system.

In considering the question of ML-style shallow polymorphism for this calculus, we discovered that the naïve generalisation of our simple type system causes an unsoundness, which is related to the unsoundness of the original ML type system when extended with imperative features and control operators. We believe that in the sequent calculus setting the exact cause of the unsoundness is clearer to see; it manifests itself as an interplay between the left propagation of cuts and the implicit polymorphic generalisation steps which are allowed in the type system. Having thus pinpointed the exact cause, we were able to define an improved type system which neatly evades the problem, while maintaining a reasonably strong facility for polymorphism. Furthermore, in the context of the classical sequent calculus, it seems natural to view the *existential* quantifier as having a dual role to that of the universal, and we show that we can define this alternative kind of polymorphic type system in a straightforward manner, by exploiting the symmetries of the underlying logic.

We succeeded in proving that our proposed shallow polymorphic type system is sound, and has a principal types property similar to that of ML. Because of the nature of our restrictions on the naïve type system, the principal types property in the setting of the $\mathcal{X}^i$-calculus was a non-trivial generalisation. In particular, we needed to introduce an extended notion of unification, to handle the combination of generic types (or type schemes) in the most general way. As a by-product, we have given formal proofs of the soundness and completeness of this operation, which we do not believe to exist in the existing literature.

In the context of classical natural deduction we have argued that a 'canonical' set of reductions, analogous with the cut elimination in classical sequent calculus, has not been previously defined. Furthermore, the calculi presented in the literature which are based on classical natural deduction, tend not to reflect Gentzen's original formulation. We believe this is largely because of the expectation of confluence which, until relatively recently, has pervaded the work on applicative-style calculi. Since, if one wishes to obtain confluence, it is necessary to restrict the 'natural' notion of reductions in a classical logic setting, a wide number of different calculi have been proposed, differing in subtle ways from one another, and none standing out as a canonical basis for the study of this paradigm. As a consequence of our desire to faithfully inhabit the original logic, and to provide a notion of reduction as general as that provided by Gentzen's cut elimination, we discovered an

extension of the famous $\lambda\mu$-calculus which we dubbed $\nu\lambda\mu$, and believe forms a simple basis for further work. The constructs of the calculus can be understood relatively intuitively, and the reductions of the calculus can be seen to incorporate a natural behaviour strongly related to that of delimited control operators. Exploring the consequences of this observation in more detail, we have shown that the historically accepted computational counterpart of the double-negation elimination inference rule of classical natural deduction (Felleisen's $\mathcal{C}$ operator) is *not* the most natural candidate, and that the reduction behaviours and typings of delimited control operators such as his $\mathcal{F}$ operator make these more natural choices.

The $\nu\lambda\mu$-calculus includes reduction behaviour which is closely related with delimited control operators, and is also able to encode many existing calculi related to classical logic. In particular, we have shown that the $\mathcal{X}^i$-calculus can be encoded into $\nu\lambda\mu$, directly preserving reductions (and typings). We believe this to be the first time that a general notion of cut elimination for classical logic has been encoded into a notion of proof normalisation for a Gentzen-style classical natural deduction. In the other direction, we were not able to provide similar results, since the reductions of $\nu\lambda\mu$ appear to be *more* general than those which are present in the $\mathcal{X}^i$-calculus. In particular, it appears that some $\nu\lambda\mu$-reductions do not naturally correspond to cut elimination steps, but instead to different kinds of sound transformations on sequent proofs (which nonetheless appear in some sense to simplify proofs).

Finally, we have been able to use the knowledge gained about encodings between the two paradigms of sequent calculus and natural deduction, to adapt the results of previous chapters to different settings. In particular, we have derived what appears to be a natural formulation of a shallow polymorphic type system for $\nu\lambda\mu$, which we believe to have the same desirable results as those we proved for the $\mathcal{X}^i$-calculus. The potential unsoundness in a shallow polymorphic type system based around classical logic was (we argue) easiest to understand directly in the setting of classical sequent calculus, and the adaptation of our solution to the natural deduction paradigm was made easy by our knowledge of encodings. We believe it would have been significantly harder to discover the same solution directly for the $\nu\lambda\mu$-calculus.

8.1 Future Work

The most obvious area for future work is the $\nu\lambda\mu$-calculus, for which we do not yet have a proof of strong normalisation. We regard this as an essential property for such a calculus, but it seems that standard techniques such as symmetric reducibility candidates are

not directly applicable, because of the generality of the reductions and the type-language. Nonetheless, we believe that such a proof should be possible, perhaps given some extension of the usual approach.

Through our attempts to encode the $\nu\lambda\mu$-calculus back to the $\mathcal{X}^i$-calculus, we have identified some possible additional reduction rules which could be added to $\mathcal{X}^i$. These rules are not cut elimination rules, but nonetheless appear to have a reasonable behaviour in terms of simplifying sequent proofs in other ways. Whether or not versions of these rules can be added to the $\mathcal{X}^i$-calculus without breaking strong normalisation is not entirely clear, but this question appears to be tied up with a strong normalisation result for $\nu\lambda\mu$, as well as with our desire for encodings in both directions between the two paradigms.

Although we have made some headway in relating the $\nu\lambda\mu$ to the field of control operators, it would be interesting to study practical applications of our calculi. There are various interesting directions which have been partially explored in the field. In [2], Ariola et. al. show that a variant of the $\overline{\lambda}\mu\tilde{\mu}$-calculus is suitable for representing the details of typical abstract machines for programming calculi, not only in terms of expressiveness, but also efficiency. This is an interesting application of the classical sequent calculus to practical programming. Ohori [64] also shows that a Curry-Howard correspondence can be established between a variant of sequent calculus and a low-level language for machine code.

In [12], Barbanera et. al. analyse the non-confluence of the symmetric λ-calculus, and relate the non-deterministic aspects to concurrent programming. In particular, they demonstrate that certain programs in their calculus can be regarded as communicating concurrent processes. It would be interesting to analyse this idea in the context of (for example) the $\mathcal{X}^i$-calculus. Recently, it has been shown that the $\mathcal{X}$-calculus (whose reductions are finer-grained than those of the $\mathcal{X}^i$-calculus), can be encoded into the π-calculus, preserving reductions [97]. This seems an interesting area for future research, since in the realm of process calculi the non-deterministic aspects of classical logic are desirable. It would be particularly interesting is to see if an encoding in the other direction were possible, i.e., to model a process calculus in a calculus based on classical logic.

In the realm of conventional programming our work on shallow polymorphism has potential practical benefits. In particular, the fact that we have identified the potentially equal status of *both* universal and existential quantification in such a type system seems to be a new idea. It is precisely the classical features of these calculi which make this the case, and most interestingly, permit examples which can only be typed using a combination of these two types of polymorphism. We believe that a sound type system including the two quantifiers together could be defined, although it seems that the question of principal types would need extensive extra work, and may even become impossible with this

extension.

8.2 Closing Remarks

Throughout this thesis we have justified our work by comparisons with other calculi and languages, however, there is no doubt that aesthetic and philosophical considerations have had a significant influence on most of our work. The original Curry-Howard correspondence is remarkably clean, and presents the well-understood λ-calculus as the computational counterpart of a canonical presentation of minimal logic. While we believe that practical considerations should not be forgotten, we find that the computational content of classical logic is interesting in its own right, and to define calculi with a similarly clean correspondence with Gentzen's original logics is an exciting achievement. Furthermore, by maintaining sufficiently general notions of reduction, we are able to represent most other calculi which exist in the field in a natural way and relate our calculi which are rooted firmly in Gentzen's work to programming concepts which are already prevalent in theoretical computer science.

Bibliography

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams Iv, D. P. Friedman, E. Kohlbecker, G. L. Steele, Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised report on the algorithmic language scheme. *Higher Order Symbolic Computation*, 11(1):7–105, 1998.

[2] Zena M. Ariola, Aaron Bohannon, and Amr Sabry. Sequent calculi and abstract machines. *ACM Transactions on Programming Languages and Systems*, 2008. to appear.

[3] Zena M. Ariola and Hugo Herbelin. Minimal classical logic and control operators. In *Proc. ICALP '03*, volume 2719 of *Lecture Notes in Computer Science*, pages 871–885. Springer, 2003.

[4] Zena M. Ariola and Hugo Herbelin. Control reduction theories: the benefit of structural substitution. *Journal of Functional Programming*, 2007. to appear.

[5] Zena M. Ariola, Hugo Herbelin, and Amr Sabry. A type-theoretic foundation of continuations and prompts. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on functional programming*, pages 40–53, New York, NY, USA, 2004. ACM.

[6] Zena M. Ariola, Hugo Herbelin, and Amr Sabry. A proof-theoretic foundation of abortive continuations. *Higher Order Symbolic Computation*, 20(4):403–429, 2007.

[7] Zena M. Ariola, Hugo Herbelin, and Amr Sabry. A type-theoretic foundation of delimited continuations. *Higher Order and Symbolic Computation*, 2007. online at http://dx.doi.org/10.1007/s10990-007-9006-0, otherwise to appear.

[8] P. Audebaud. Explicit substitutions for the lambda-mu calculus, 1994. Available at `http://citeseer.ist.psu.edu/audebaud94explicit.html`.

[9] P. Audebaud and S. van Bakel. Understanding $\mathcal{X}$ with $\lambda\mu$. Consistent interpretations of the implicative sequent calculus in natural deduction. Available from `http://www.doc.ic.ac.uk/~svb/Research/Papers/AvB.pdf`, 2006.

[10] Franco Barbanera and Stefano Berardi. A symmetric lambda-calculus for classical program extraction. In *Proceedings of TACS '94*, volume 789 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[11] Franco Barbanera and Stefano Berardi. A strong normalization result for classical logic. *Annals of Pure and Applied Logic*, 76(2):99–116, 1995.

[12] Franco Barbanera, Stefano Berardi, and M. Schivalocchi. "classical" programming-with-proofs in λ_{PA}^{Sym}:an analysis of non-confluence. In *Proceedings of TACS'97*, volume 1281 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[13] Henk Barendregt and Silvia Ghilezan. Lambda terms for natural deduction, sequent calculus and cut elimination, 1998.

[14] Gilles Barthe, Fairouz Kamareddine, and Alejandro Rios. Explicit substitutions for the lambda-calculus. In *ALP/HOA*, pages 209–223, 1997.

[15] Paul Beame and Toniann Pitassi. Propositional proof complexity: past, present, and future. In *Current trends in theoretical computer science: entering the 21st century*, pages 42–70, River Edge, NJ, USA, 2001. World Scientific Publishing Co., Inc.

[16] Josh Berdine, Peter O'hearn, Uday Reddy, and Hayo Thielecke. Linear continuation-passing. *Higher Order Symbolic Computation*, 15(2-3):181–208, 2002.

[17] G. M. Bierman. A computational interpretation of the $\lambda\mu$-calculus. In *Proceedings of Symposium on Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*, pages 336–345. Springer-Verlag, 1998.

[18] R. Bloo and K.H. Rose. Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. In *CSN'95 – Computer Science in the Netherlands*, pages 62–72, 1995.

[19] B. R. Boričić. On sequence-conclusion natural deduction systems. *Journal of Philosophical Logic*, 14:359–377, 1985.

[20] A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33:346–366, 1932. Second paper with same title in Vol. 33, pages 839–864, of same journal.

[21] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proceedings of ICFP'00*, pages 233–243. ACM, 2000.

[22] Haskell B. Curry. Functionality in Combinatory Logic. In *Proceedings of National Academy of Sciences, U.S.A.*, volume 20, pages 584–590, 1934.

[23] Haskell B. Curry and Robert Feys. *Combinatory Logic Vol. I.* North-Holland, Amsterdam, 1958.

[24] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings 9$^{\text{th}}$ ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[25] Olivier Danvy and Andrzej Filinski. Abstracting control. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160, New York, NY, USA, 1990. ACM.

[26] Philippe de Groote. A cps-translation of the lambda-μ-calculus. In *CAAP '94: Proceedings of the 19th International Colloquium on Trees in Algebra and Programming*, pages 85–99, London, UK, 1994. Springer-Verlag.

[27] Philippe de Groote. On the relation between the lambda-mu-calculus and the syntactic theory of sequential control. In *LPAR '94: Proc. 5th International Conference on Logic Programming and Automated Reasoning*, pages 31–43, London, UK, 1994. Springer-Verlag.

[28] Philippe de Groote. Strong normalization of classical natural deduction with disjunction. In *TLCA*, pages 182–196, 2001.

[29] D. Dougherty, S. Ghilezan, and P. Lescanne. Intersection and Union Types in the $\overline{\lambda}\mu\tilde{\mu}$-calculus. In *Electronic Proceedings of 2nd International Workshop* Intersection Types and Related Systems *(ITRS'04), Turku, Finland*, Electronic Notes in Theoretical Computer Science, 2004.

[30] D. Dougherty, S. Ghilezan, and P. Lescanne. Characterizing strong normalization in the Curien-Herbelin symmetric lambda calculus: extending the Coppo-Dezani heritage. *Theoretical Computer Science*, 398, 2008. Coppo, Dezani, and Ronchi Festschrift.

[31] A G Dragalin. Mathematical intuitionism: Introduction to proof theory, volume 67 of translations of mathematical monographs. *of Translations of Mathematical Monographs. American Mathematical Society*, 67, 1988.

[32] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ml. In *Proceedings of POPL '91*, pages 163–173, New York, NY, USA, 1991. ACM Press.

[33] R. Kent Dybvig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 2006. To appear.

[34] Roy Dyckhoff and Luis Pinto. Cut-elimination and a permutation-free sequent calculus for intuitionistic logic. *Studia Logica*, 60(1):107–118, 1998.

[35] Roy Dyckhoff and Luís Pinto. Permutability of proofs in intuitionistic sequent calculi. *Theoretical Computer Science*, 212(1–2):141–155, 1999.

[36] M. Felleisen. A historical note: On the indiana control operators, 2007. Appendix to [4].

[37] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Journal of Theoretical Computer Science*, 52:205–237, 1987.

[38] Matthias Felleisen. The theory and practice of first-class prompts. In *POPL*, pages 180–190, 1988.

[39] Gerhard Gentzen. Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

[40] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[41] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse et son application l'élimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the Second Scandinavian Logic Symposium (Univ. Oslo, Oslo, 1970)*, volume 63, pages 63–92. North-Holland, 1971.

[42] Jean-Yves Girard. Linear logic. *Journal of Theoretical Computer Science*, 50(1):1–102, 1987.

[43] T. Griffin. A formulae-as-types notion of control. In *Conf. Record 17th Annual ACM Symp. on Principles of Programming Languages, POPL'90*, pages 47–57. ACM Press, 1990.

[44] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ml-like languages. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 12–23, New York, NY, USA, 1995. ACM.

[45] Robert Harper and Mark Lillibridge. Ml with callcc is unsound. Message sent to the "sml" mailing list.

[46] Hugo Herbelin. A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 1995.

[47] Hugo Herbelin and Silvia Ghilezan. An approach to call-by-name delimited continuations. *SIGPLAN Not.*, 43(1):383–394, 2008.

[48] R. Hieb and R. Kent Dybvig. Continuations and concurrency. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 128–136, New York, NY, USA, 1990. ACM.

[49] J.R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[50] W. A. Howard. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[51] Jean-Baptiste Joinet. *Étude de la Normalisation du Calcul des Séquents Classique à Travers la Logique Linéaire*. PhD thesis, Université Paris 7, 1993.

[52] Andrew Kennedy. Compiling with continuations, continued. *SIGPLAN Not.*, 42(9):177–190, 2007.

[53] O. Kiselyov. How to remove a dynamic prompt: static and dynamic delimited continuation operators are equally expressible. Technical report, Dept. Computer Science, Indiana University, 2005.

[54] S.C. Kleene. *Introduction to Metamathematics*. North-Holland, 1952.

[55] F. Lamarche and L. Straßburger. Naming proofs in classical propositional logic. *Typed Lambda Calculi and Applications*, 3461:246–261, 2005.

[56] Stéphane Lengrand. Call-by-value, call-by-name, and strong normalization for the classical sequent calculus. In *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.

[57] Xavier Leroy. Polymorphism by name for references and continuations. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–231, New York, NY, USA, 1993. ACM.

[58] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[59] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1990.

[60] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[61] C.R. Murthy. An evaluation semantics for classical proofs. *Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 96–107, July 1991.

[62] C.R. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In *Stanford University*, pages 49–71, 1992.

[63] Ichiro Ogata. Gentzen-style classical proofs as lambda-mu terms. In *In Proceedings of the Asian Computing Science Conference 99*, pages 266–280, 1999.

[64] Atsushi Ohori. A proof theory for machine code. *ACM Transactions on Programming Languages and Systems*, 29(6):36, 2007.

[65] C.-H. Luke Ong and Charles A. Stewart. A Curry-Howard foundation for functional computation with control. In *Proceedings of 24th Symposium on Principles of Programming Languages*, pages 215–227. ACM Press, New York, 1997.

[66] M. Parigot. An algorithmic interpretation of classical natural deduction. In *Proceedings of LPAR'92*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer-Verlag, 1992.

[67] M. Parigot. Proofs of strong normalisation for second order classical natural deduction. *Journal of Symbolic Logic*, 62(4):1461–1479, December 1997.

[68] Michel Parigot. Church-rosser property in classical free deduction. In *Papers presented at the second annual Workshop on Logical environments*, pages 273–296, New York, NY, USA, 1993. Cambridge University Press.

[69] Michel Parigot. Classical proofs as programs. In *KGC '93: Proceedings of Third Kurt Gödel Colloquium on Computational Logic and Proof Theory*, pages 263–276, London, UK, 1993. Springer-Verlag.

[70] Michel Parigot. On the computational interpretation of negation. In *Proceedings of 14th Annual Conference of the EACSL on Computer Science Logic*, pages 472–484, London, UK, 2000. Springer-Verlag.

[71] Garrel Pottinger. Normalization as a homomorphic image of cut-elimination. *Annals of Mathematical Logic*, 12:323–357, 1977.

[72] Dag Prawitz. *Natural Deduction, A Proof-Theoretical Study*. Almqvist and Wiskell, Stockholm, 1965.

[73] W. Py. *Confluence en $\lambda\mu$-calcul*. PhD thesis, Université de Savoie, 1998.

[74] Jayshan Raghunandan. *Curry-Howard Calculi from Classical Logical Connectives: A Generic Tool for Higher-Order Term Graph Rewriting*. PhD thesis, Imperial College London, 2009.

[75] Jayshan Raghunandan and Alexander J. Summers. On the computational representation of classical logical connectives. *Electronic Notes in Theoretical Computer Science*, 171(3):85–109, 2007.

[76] Stephen Read. Harmony and autonomy in classical logic. *Journal of Philosophical Logic*, 29:123–154, 2000.

[77] N. Rehof and M. Sørensen. The $\lambda\delta$ calculus. *Theoretical Aspects of Computer Software*, 789:516–542., 1994.

[78] J.C. Reynolds. Towards a Theory of Type Structures. In B. Robinet, editor, *Proceedings of 'Colloque sur la Programmation',* Paris, France, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[79] Edmund Robinson. Proof nets for classical logic. *Journal of Logic and Computation*, 13(5):777–797, 2003.

[80] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[81] Jérôme Rocheteau. $\lambda\mu$-Calculus and Duality: Call-by-Name and Call-by-Value. In Jrgen Giesl, editor, *Term Rewriting and Applications*, volume 3467 of *Lecture Notes in Computer Science*, pages 204–218. Springer-Verlag, March 2005.

[82] José Espírito Santo. Revisiting the correspondence between cut elimination and normalisation. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 600–611. Springer-Verlag, 2000.

[83] José Espírito Santo. An isomorphism between a fragment of sequent calculus and an extension of natural deduction. In *LPAR '02: Proceedings of the 9th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 352–366, London, UK, 2002. Springer-Verlag.

[84] José Espírito Santo. Completing herbelin's programme. *Typed Lambda Calculi and Applications*, 4583/2007:118–132, 2007.

[85] Chung-chieh Shan. Shift to control. Proceedings of the 5th workshop on Scheme and functional programming 600, Indiana University, 2004.

[86] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Proceedings of the 5th workshop on Scheme and functional programming*, 3(1):67–99, January 1990.

[87] Kristian Støvring and Soren B. Lassen. A complete, co-inductive syntactic theory of sequential control and state. *SIGPLAN Not.*, 42(1):161–172, 2007.

[88] Thomas Streicher and Bernhard Reus. Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6):543–572, 1998.

[89] Alexander J. Summers and Steffen van Bakel. Approaches to polymorphism in classical sequent calculus. In *ESOP*, pages 84–99, 2006.

[90] W. W. Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32(2):187–199, 1967.

[91] Mads Tofte. Type inference for polymorphic references. *Journal of Information and Computation*, 89(1):1–34, 1990.

[92] Christian Urban. *Classical Logic and Computation*. PhD thesis, University of Cambridge, October 2000.

[93] H. Schellinx V. Danos, J.-B. Joinet. Lkt and lkq: sequent calculi for second order logic based upon dual linear decomposition of classical implication. *Advances in Linear Logic*, pages 43–59, 1995.

[94] S. van Bakel. Completeness and partial soundness results for intersection & union typing for $\overline{\lambda}\mu\tilde{\mu}$. Submitted, 2008.

[95] S. van Bakel. Reduction in $\mathcal{X}$ does not agree with Intersection and Union Types. In *Electronic Proceedings of 4th International Workshop* Intersection Types and Related Systems *(ITRS'08), Turin, Italy*, 2008.

[96] S. van Bakel. Subject reduction vs intersection / union types in $\overline{\lambda}\mu\tilde{\mu}$ (extended abstract). In *Proceedings of BCS International Academic Conference* Visions of Computer Science*, London, England*, pages 249–258, 2008.

[97] S. van Bakel, L. Cardelli, and M.G. Vigliotti. From $\mathcal{X}$ to π: Representing classical sequent calculus in π-calculus. In *Second International Workshop on Classical Logic and Computation*, 2008.

[98] S. van Bakel, S. Lengrand, and P. Lescanne. The language $\mathcal{X}$: circuits, computations and classical logic. In *Proceedings of ICTCS'05*, volume 3701 of *Lecture Notes in Computer Science*, pages 66–80. Springer-Verlag, 2005.

[99] S. van Bakel and J. Raghunandan. Implementing $\mathcal{X}$. In *International Workshop on Term Graph Rewriting*, 2004.

[100] Jan von Plato. Natural deduction with general elimination rules. *Archive for Mathematical Logic*, 40(7):541–567, 2001.

[101] Jan von Plato. Gentzen's proof of normalisation for natural deduction. *Bulletin of Symbolic Logic*, 14(2):240–257, 2008.

[102] Philip Wadler. Call-by-value is dual to call-by-name - reloaded. In *RTA*, pages 185–203, 2005.

[103] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.

[104] J. B. Wells. The essence of principal typings. In *Proceedings of 29th International Colloquium on Automata, Languages, and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 913–925. Springer-Verlag, 2002.

[105] Andrew Wright. Polymorphism for imperative languages without imperative types. Technical Report TR93-200, College of Computer and Information Science, Northeastern University, 21, 1993.

[106] Yoriyuki Yamagata. Strong normalization of second order symmetric lambda-mu calculus. *Lecture Notes in Computer Science*, 2215:459–??, 2001.

[107] Yoriyuki Yamagata. Strong normalization of a symmetric lambda calculus for second-order classical logic. *Archive for Mathematical Logic*, 41(1):91–99, 2002.

[108] Jeffery Zucker. The correspondence between cut-elimination and normalization, part i and ii. *Annals of Mathematical Logic*, 7:1–112,113–155, 1974.

Appendix A

Supporting Proofs

A.1 Proofs for Chapter 3

Proof A.1.1 (of Theorem 3.3.12)**.**

1. *Soundness: By induction on the structure of the term,* P.

   ($P \equiv \langle x.\alpha \rangle$**):** *Then* $\Gamma = \{x{:}\varphi\}$ *and* $\Delta = \{\alpha{:}\varphi\}$ *for some fresh* φ*. By the rule* (ax)*, we obtain* $\langle x.\alpha \rangle \mathrel{:\cdot} \{x{:}\varphi\} \vdash \{\alpha{:}\varphi\}$

   ($P \equiv \widehat{x}Q\widehat{\alpha}\cdot\beta$**):** *Then* $\Gamma = (S\ (\Gamma_Q \backslash x))$ *and* $\Delta = (S\ (\beta{:}C, \Delta_Q \backslash \alpha))$*. By induction, we obtain* $Q \mathrel{:\cdot} \Gamma_Q \vdash \Delta_Q$*.*

   Notice that, since $A =$ *typeof* $x\ \Gamma$*, either* $x : A \in \Gamma_Q$ *or* $x \notin \Gamma_Q$*. Hence* $(\Gamma_Q \backslash x), x{:}A \supseteq \Gamma_Q$*. By similar argument,* $\alpha{:}B, (\Delta_Q \backslash \alpha) \supseteq \Delta_Q$*. Then, by weakening where necessary, we have* $Q \mathrel{:\cdot} (\Gamma_Q \backslash x), x{:}A \vdash \alpha{:}B, (\Delta_Q \backslash \alpha)$*.*

   Applying the rule $(\rightarrow\mathcal{R})$*, we obtain* $\widehat{x}Q\widehat{\alpha}\cdot\beta \mathrel{:\cdot} \Gamma_Q \backslash x \vdash \beta{:}A{\rightarrow}B, \Delta_Q \backslash \alpha$*. Now, by Lemma 3.3.10,* $\widehat{x}Q\widehat{\alpha}\cdot\beta \mathrel{:\cdot} (S\ (\Gamma_Q \backslash x)) \vdash (S\ (\beta{:}A{\rightarrow}B, \Delta_Q \backslash \alpha))$*. Notice that* $(S\ (\Gamma_Q \backslash x)) = \Gamma$*. By definition of* S*,* $(S\ A{\rightarrow}B) = (S\ C)$*, and so we also have* $(S\ (\beta{:}A{\rightarrow}B, \Delta_Q \backslash \alpha)) = \Delta$

   ($P \equiv Q\widehat{\alpha}\,[y]\,\widehat{x}R$**):** *By definition, we have* $\Gamma = (S_3\ (S_2 \circ S_1\ (\Gamma_Q \cup (\Gamma_R \backslash x))), y{:}C)$ *and* $\Delta = (S_3 \circ S_2 \circ S_1\ ((\Delta_Q \backslash \alpha) \cup \Delta_R))$*. By induction, twice,* $Q \mathrel{:\cdot} \Gamma_Q \vdash \Delta_Q$ *and* $R \mathrel{:\cdot} \Gamma_R \vdash \Delta_R$*.*

   By weakening we have $Q \mathrel{:\cdot} \Gamma_Q \vdash (\Delta_Q \backslash \alpha), \alpha{:}A$ *and* $R \mathrel{:\cdot} x{:}B, (\Gamma_R \backslash x) \vdash \Delta_R$*. Let* $S = S_3 \circ S_2 \circ S_1$*, for brevity.*

   By Lemma 3.3.10, we know that both $Q \mathrel{:\cdot} (S\ \Gamma_Q) \vdash (S\ ((\Delta_Q \backslash \alpha), \alpha{:}A))$ *and* $R \mathrel{:\cdot} (S\ (x{:}B, (\Gamma_R \backslash x))) \vdash (S\ \Delta_R)$*.*

By definition of S_1, we know that $(S\ (\Gamma_Q \cup (\Gamma_R \backslash x)))$ gives a well-formed context. Since x appears as a binder in $Q\widehat{\alpha}\,[y]\,\widehat{x}R$, we may assume that x does not also appear free in the term. In particular, we can assume that $x \notin \Gamma_Q$, and therefore that $(S\ (\Gamma_Q \cup (\Gamma_R \backslash x), x{:}B))$ is a well-formed context. By similar argument, given the definition of S_2, we have that $(S\ (\alpha{:}A, (\Delta_P \backslash \alpha) \cup \Delta_Q))$ is well-formed.

By further weakening of the judgements, and applying the rule $(\rightarrow\mathcal{L})$, we obtain $Q\widehat{\alpha}\,[y]\,\widehat{x}R :\cdot (S\ (\Gamma_Q \cup (\Gamma_R \backslash x))), y{:}(S\ A) \rightarrow (S\ B) \vdash (S\ (\Delta_P \backslash \alpha) \cup \Delta_Q)$. Notice that $\Delta = (S\ ((\Delta_P \backslash \alpha) \cup \Delta_Q))$. Since $S_3 = unify\ C\ (S_2 \circ S_1\ A \rightarrow B)$, we have $(S\ A) \rightarrow (S\ B) = (S_3\ C)$. Hence, $(S\ (\Gamma_Q \cup (\Gamma_R \backslash x))), y{:}(S\ A) \rightarrow (S\ B) = \Gamma$.

($P \equiv Q\widehat{\alpha} \dagger \widehat{x}R$): *Similar to the previous case.*

($P \equiv \widehat{x}Q \cdot \alpha$): *Similar to the case ($P \equiv \widehat{x}Q\widehat{\alpha} \cdot \beta$).*

($P \equiv x \cdot Q\widehat{\alpha}$): *Similar to the case ($P \equiv \widehat{x}Q\widehat{\alpha} \cdot \beta$).*

2. *Completeness: By induction on the structure of the term, P.*

   ($P \equiv \langle x.\alpha \rangle$): *Then $\Gamma = \Gamma', x{:}A$ and $\Delta = \alpha{:}A, \Delta'$ for some type A and contexts Γ', Δ'. We have $pC(\langle x.\alpha \rangle) = \langle x{:}\varphi; \alpha{:}\varphi \rangle$ for a fresh type-variable φ. Take S to be the substitution $(\varphi \mapsto A)$.*

   $P \equiv \widehat{x}Q\widehat{\alpha} \cdot \beta$ *Then say $\Delta = \Delta', \beta{:}D \rightarrow E$. From the $(\rightarrow\mathcal{R})$ rule, we obtain that $Q :\cdot \Gamma, x{:}D \vdash \alpha{:}E, \Delta'$.*

   By induction, there exists S_1 such that $(S_1\ \Gamma_Q) \subseteq \Gamma, x{:}D$ and $(S_1\ \Delta_Q) \subseteq \alpha{:}E, \Delta'$. Define the substitutions S_2, S_3 and S_4 as follows:

$$S_2 = \begin{cases} id & \text{if } x{:}A \in \Gamma_Q \\ (A \mapsto D) & otherwise \end{cases}$$

$$S_3 = \begin{cases} id & \text{if } \alpha{:}B \in \Delta_Q \\ (B \mapsto E) & otherwise \end{cases}$$

$$S_4 = \begin{cases} id & \text{if } \beta{:}C \in \Delta_Q \\ (C \mapsto (D \rightarrow E)) & otherwise \end{cases}$$

   Note that S_2,S_3,S_4 act on fresh type variables (if any). Let $S' = S_4 \circ S_3 \circ S_2 \circ S_1$. Claim:

   (i) $(S'\ A) = D$

   (ii) $(S'\ \Gamma_Q \backslash x{:}A) \subseteq \Gamma$

(iii) $(S'\ B) = E$

(iv) $(S'\ \Delta_Q \backslash \alpha{:}B) \subseteq \Delta'$

(v) $(S'\ C) = (S'\ A{\rightarrow}B)$

(vi) $(S'\ \beta{:}C, \Delta_Q \backslash \alpha{:}B) \subseteq \Delta', \beta{:}D{\rightarrow}E$

Proof. **(i) and (ii):** Consider two cases:

$x{:}A \in \Gamma_Q$:

Then, since $(S_1\ \Gamma_Q) \subseteq \Gamma, x{:}D$ we must have $(S_1\ A) = D$. Notice that $S_2 = id$ in this case, and that S_3,S_4 act on other type variables (if any). Hence $(S'\ A) = (S_1\ A) = D$ (i).

In this case, $\Gamma_Q = (\Gamma_Q \backslash x{:}A), x{:}A$. We have:

$$\begin{aligned}(S_1\ \Gamma_Q \backslash x{:}A), x{:}D &= (S_1\ \Gamma_Q \backslash x{:}A), x{:}(S_1\ A) \\ &= (S_1\ (\Gamma_Q \backslash x{:}A), x{:}A) \\ &= (S_1\ \Gamma_Q) \\ &\subseteq \Gamma, x{:}D\end{aligned}$$

Therefore $(S_1\ \Gamma_Q \backslash x{:}A) \subseteq \Gamma$ (ii).

$x{:}A \notin \Gamma_Q$:

Then A is fresh, $(S_1\ A) = A$ and $S_2 = (A{\mapsto}D)$. Hence $(S'\ A) = D$ (i).

Since $x{:}A \notin \Gamma_Q$, we have $\Gamma_Q \backslash x{:}A = \Gamma_Q$ and also $x \notin (S_1\ \Gamma_Q)$. Therefore:

$$\begin{aligned}(S_1\ \Gamma_Q) \subseteq \Gamma, x{:}D &\Rightarrow (S_1\ \Gamma_Q) \subseteq \Gamma \\ &\Rightarrow (S_1\ \Gamma_Q \backslash x{:}A) \subseteq \Gamma \\ &\Rightarrow (S'\ \Gamma_Q \backslash x{:}A) \subseteq \Gamma\end{aligned}$$

as required (ii).

(iii) and (iv): By similar argument to (i) and (ii), considering cases for $\alpha{:}B \in \Delta_Q$.

(v): Consider two cases:

$\beta{:}C \in \Delta_Q$:

We have by part (iv) that $(S'\ \Delta_Q \backslash \alpha{:}B) \subseteq \Delta'$, and so in particular, that $\beta{:}(S'\ C) \in \Delta'$. However, we have $\widehat{x}Q\widehat{\alpha}{\cdot}\beta :\cdot\ \Gamma \vdash \Delta', \beta{:}D{\rightarrow}E$, and so $\Delta', \beta{:}D{\rightarrow}E$ must be a well-formed context. Hence we must have:

$$\begin{aligned}(S'\ C) &= D{\rightarrow}E \\ &= (S'\ A){\rightarrow}(S'\ B) \\ &= (S'\ A{\rightarrow}B)\end{aligned}$$

$\beta{:}C \notin \Delta_Q$:

Then C is fresh, and $S_4 = (C \mapsto (D \to E))$. Hence $(S'\ C) = D \to E = (S'\ A \to B)$.

(vi): By (v), $(S'\ C) = D \to E$. Hence this follows directly from (iv).

□

Since S' unifies C and $A \to B$, by Lemma 3.3.9(i) we have $\exists S$ such that $S' = S \circ S_u$ (where S_u = unify C $A \to B$, as above). Now, letting $\langle \Gamma_P; \Delta_P \rangle$ = pC (P), by the definition of the algorithm, $\Gamma_P = (S_u\ \Gamma_Q \backslash x{:}A)$ and $\Delta_P = (S_u\ \beta{:}C, \Delta_Q \backslash \alpha{:}B)$. Using the Claim (ii) above we have:

$$\begin{aligned}(S\ \Gamma_P) &= (S\ S_u\ \Gamma_Q \backslash x{:}A) \\ &= (S'\ \Gamma_Q \backslash x{:}A) \\ &\subseteq \Gamma\end{aligned}$$

Similarly, by the Claim (vi) above, $(S\ \Delta_P) \subseteq \Delta', \beta{:}D \to E = \Delta$, as required.

($P \equiv Q\widehat{\alpha}\,[y]\,\widehat{x}R$): *Then, by the rule $(\to\mathcal{L})$, we have $\Gamma = \Gamma', y{:}D \to E$ for some types D, E and context Γ'. From the rule, we know $Q\widehat{\alpha}\,[y]\,\widehat{x}R :\cdot \Gamma', y{:}D \to E \vdash \Delta$ and $Q :\cdot \Gamma' \vdash \alpha{:}D, \Delta$ and $R :\cdot \Gamma', x{:}E \vdash \Delta$.*

Let

$$\begin{aligned}\langle \Gamma_Q; \Delta_Q \rangle &= pc(Q) \\ \langle \Gamma_R; \Delta_R \rangle &= pc(R) \\ A &= \textit{typeof}\ \alpha\ \Delta_Q \\ B &= \textit{typeof}\ x\ \Gamma_R \\ S_\Gamma &= \textit{unifyContexts}\ \Gamma_Q\ (\Gamma_R \backslash x) \\ S_\Delta &= \textit{unifyContexts}\ (S_\Gamma\ \Delta_Q \backslash \alpha)\ (S_\Gamma\ \Delta_R) \\ C &= \textit{typeof}\ y\ (S_\Delta \circ S_\Gamma\ \Gamma_Q \cup (\Gamma_R \backslash x)) \\ S_C &= \textit{unify}\ C\ (S_\Delta \circ S_\Gamma\ A \to B)\end{aligned}$$

By induction, twice, $\exists S_1, S_2$ such that:

$$\begin{aligned}(S_1\ \Gamma_Q) &\subseteq \Gamma' \\ (S_1\ \Delta_Q) &\subseteq \alpha{:}D, \Delta \\ (S_2\ \Gamma_R) &\subseteq \Gamma', x{:}E \\ (S_2\ \Delta_R) &\subseteq \Delta\end{aligned}$$

Note that we may assume that the sets of type variables occurring in each of the two pairs $\langle \Gamma_Q; \Delta_Q \rangle$ and $\langle \Gamma_R; \Delta_R \rangle$ are distinct (since they should all be fresh at some stage in the calls). This means in particular we may assume that S_1 has no effect on $\langle \Gamma_R; \Delta_R \rangle$ and likewise for S_2, $\langle \Gamma_Q; \Delta_Q \rangle$. Define the

substitutions S_3, S_4 and S_5 by:

$$S_3 = \begin{cases} id & \textit{if } \alpha{:}A \in \Delta_Q \\ (A \mapsto D) & otherwise \end{cases}$$

$$S_4 = \begin{cases} id & \textit{if } x{:}B \in \Gamma_R \\ (B \mapsto E) & otherwise \end{cases}$$

$$S_5 = \begin{cases} id & \textit{if } y{:}C \in (S_\Delta \circ S_\Gamma\ \Gamma_Q \cup (\Gamma_R \backslash x)) \\ (C \mapsto (D \rightarrow E)) & otherwise \end{cases}$$

Let $S' = S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1$.
Claim:

(i) $(S'\ B) = E$

(ii) $(S'\ \Gamma_Q \cup (\Gamma_R \backslash x{:}B)) \subseteq \Gamma'$

(iii) $(S'\ A) = D$

(iv) $(S'\ (\Delta_Q \backslash \alpha{:}A) \cup \Delta_R) \subseteq \Delta$

(v) *If* $y{:}C \in (S_\Delta \circ S_\Gamma\ \Gamma_Q \cup (\Gamma_R \backslash x))$: *then* $\forall C', (y{:}C' \in \Gamma_Q \cup (\Gamma_R \backslash x{:}B) \Rightarrow (S'\ C') = S'\ A \rightarrow B)$

(vi) *If* $y{:}C' \notin \Gamma_Q \cup (\Gamma_R \backslash x)$, *then* $(S'\ C) = (S'\ A \rightarrow B)$.

Proof. **(i) and (ii):** Consider two cases:

$x{:}B \in \Gamma_R$:

Since $(S_2\ \Gamma_R) \subseteq \Gamma', x{:}E$ we must have $(S_2\ B) = E$. Hence $(S'\ B) = E$ (i).

In this case, $\Gamma_R = (\Gamma_R \backslash x{:}B), x{:}B$. We have $(S_2\ (\Gamma_R \backslash x{:}B), x{:}B) \subseteq \Gamma', x{:}E$, and so $(S_2\ \Gamma_R \backslash x{:}B) \subseteq \Gamma'$. We know also that $(S_1\ \Gamma_Q) \subseteq \Gamma'$, and so this must hold for the union of these two contexts after both substitutions have been applied: $(S'\ \Gamma_Q \cup (\Gamma_R \backslash x{:}B)) \subseteq \Gamma'$ (ii).

$x{:}B \notin \Gamma_R$:

Then $S_4 = (B \mapsto E)$ and so $(S'\ B) = E$ (i).

We have $\Gamma_R \backslash x{:}B = \Gamma_R$ and also $x \notin (S'\ \Gamma_R)$. As usual, since S_1 does not act on Γ_Q and S_3,S_4,S_5 only on fresh type-variables, we may deduce from $(S_2\ \Gamma_R) \subseteq \Gamma', x{:}E$ that $(S'\ \Gamma_R) \subseteq \Gamma', x{:}E$. By the above, we deduce $(S'\ \Gamma_R \backslash x{:}B) \subseteq \Gamma'$. We have also that $(S_1\ \Gamma_Q) \subseteq \Gamma'$, i.e., $(S'\ \Gamma_Q) \subseteq \Gamma'$ and so we may deduce that $(S'\ \Gamma_Q \cup (\Gamma_R \backslash x{:}B)) \subseteq \Gamma'$ as required (ii).

(iii) and (iv): By similar argument to (i) and (ii), considering cases for $\alpha{:}B \in \Delta_Q$.

(v): Let C' be a type satisfying $y{:}C' \in \Gamma_Q \cup (\Gamma_R \backslash x)$. Notice that such C' need not be unique (since y may occur in both Γ_Q and $(\Gamma_R \backslash x)$, but the argument which follows works for both of the occurrences, where there are two.

We have by part (ii) that $(S'\ \Gamma_Q \cup (\Gamma_R \backslash x{:}B)) \subseteq \Gamma'$, and so in particular that $y{:}(S'\ C') \in \Gamma'$. However, we have $Q\widehat{\alpha}\,[y]\,\widehat{x}R :\cdot \Gamma', y{:}D{\rightarrow}E \vdash \Delta$, and so $\Gamma', y{:}D{\rightarrow}E$ must be a well-formed context. Hence we must have:

$$\begin{aligned}(S'\ C') &= D{\rightarrow}E \\ &= (S'\ A){\rightarrow}(S'\ B) \\ &= (S'\ A{\rightarrow}B)\end{aligned}$$

(vi): In this case, C is fresh, and $S_5 = (C{\mapsto}(D{\rightarrow}E))$. Hence $(S'\ C) = D{\rightarrow}E = (S'\ A{\rightarrow}B)$ (using (i) and (iii)).

□

From here on, let C' be a type such that:

$$S_3 = \begin{cases} y{:}C' \in \Gamma_Q \cup (\Gamma_R \backslash x) & \textit{if } y{:}C \in (S_\Delta \circ S_\Gamma\ \Gamma_Q \cup (\Gamma_R \backslash x)) \\ C' = C & \textit{otherwise} \end{cases}$$

Combining (v) and (vi) above, it is clear that $(S'\ C') = (S'\ A{\rightarrow}B)$ in all cases. Furthermore, in both cases we have $(S_\Delta \circ S_\Gamma\ C') = C$ (since C is fresh in the latter case, and so the substitutions have no effect).

By (ii) it can be seen that S' unifies the contexts Γ_Q and $(\Gamma_R \backslash x{:}B)$, while from (iv) that it also unifies $(\Delta_Q \backslash \alpha{:}A)$ and Δ_R. It also unifies C' and $A{\rightarrow}B$, hence, applying Lemma 3.3.9, we have $\exists S$ such that $S' = S \circ S_C \circ S_\Delta \circ S_\Gamma$, observing that $S_C = \textit{unify}\ C\ (S_\Delta \circ S_\Gamma\ A{\rightarrow}B) = \textit{unify}\ (S_\Delta \circ S_\Gamma\ C')\ (S_\Delta \circ S_\Gamma\ A{\rightarrow}B)$.

Now, letting $\langle \Gamma_P; \Delta_P \rangle = \textbf{pC}\,(P)$, by the definition of the algorithm, $\Gamma_P = (S_C\ (S_\Delta \circ S_\Gamma\ \Gamma_Q \cup (\Gamma_R \backslash x)), y{:}C)$ and $\Delta_P = (S_C \circ S_\Delta \circ S_\Gamma\ (\Delta_Q \backslash \alpha) \cup \Delta_R)$. Using the Claim (ii) above we have:

$$\begin{aligned}(S\ \Gamma_P) &= (S\ S_C\ (S_\Delta \circ S_\Gamma\ \Gamma_Q \cup (\Gamma_R \backslash x)), y{:}C) \\ &= (S\ S_C\ (S_\Delta \circ S_\Gamma\ \Gamma_Q \cup (\Gamma_R \backslash x)), y{:}S_\Delta \circ S_\Gamma\ C') \\ &= (S\ S_C\ S_\Delta \circ S_\Gamma\ \Gamma_Q \cup (\Gamma_R \backslash x), y{:}C') \\ &= (S \circ S_C \circ S_\Delta \circ S_\Gamma\ \Gamma_Q \cup (\Gamma_R \backslash x), y{:}C') \\ &= (S'\ \Gamma_Q \cup (\Gamma_R \backslash x), y{:}C')\end{aligned}$$

Now, combining the fact that $(S'\ C') = (S'\ A{\rightarrow}B) = D{\rightarrow}E$ with (ii) above, we obtain $(S'\ \Gamma_Q \cup (\Gamma_R \backslash x), y{:}C') \subseteq \Gamma', y{:}D{\rightarrow}E$, i.e. $(S\ \Gamma_P) \subseteq \Gamma$ as required. Similarly, by the Claim (iv) above, $(S\ \Delta_P) \subseteq \Delta$, as required.

(cut) *Then* $P \equiv Q\widehat{\alpha} \dagger \widehat{x}R$. *From the rule,* $Q\widehat{\alpha} \dagger \widehat{x}R :\cdot \Gamma \vdash \Delta$ *and* $Q :\cdot \Gamma \vdash \alpha{:}D, \Delta$ *and* $R :\cdot \Gamma, x{:}D \vdash \Delta$, *for some type* D.

Let

$$\begin{aligned}
\langle \Gamma_Q ; \Delta_Q \rangle &= pc(Q) \\
\langle \Gamma_R ; \Delta_R \rangle &= pc(R) \\
A &= \textit{typeof}\ \alpha\ \Delta_Q \\
B &= \textit{typeof}\ x\ \Gamma_R \\
S_\Gamma &= \textit{unifyContexts}\ \Gamma_Q\ (\Gamma_R \backslash x) \\
S_\Delta &= \textit{unifyContexts}\ (S_\Gamma\ \Delta_Q \backslash \alpha)\ (S_\Gamma\ \Delta_R) \\
S_U &= \textit{unify}\ (S_\Delta \circ S_\Gamma\ A)\ (S_\Delta \circ S_\Gamma\ B)
\end{aligned}$$

By induction, twice, $\exists S_1, S_2$ *such that:*

$$\begin{aligned}
(S_1\ \Gamma_Q) &\subseteq \Gamma \\
(S_1\ \Delta_Q) &\subseteq \alpha{:}D, \Delta \\
(S_2\ \Gamma_R) &\subseteq \Gamma, x{:}D \\
(S_2\ \Delta_R) &\subseteq \Delta
\end{aligned}$$

Define the substitutions S_3 *and* S_4 *by:*

$$S_3 = \begin{cases} \textit{id} & \textit{if}\ \alpha{:}A \in \Delta_Q \\ (A \mapsto D) & otherwise \end{cases}$$

$$S_4 = \begin{cases} \textit{id} & \textit{if}\ x{:}B \in \Gamma_R \\ (B \mapsto D) & otherwise \end{cases}$$

Let $S' = S_4 \circ S_3 \circ S_2 \circ S_1$.

Claim:

(i) $(S'\ A) = D$

(ii) $(S'\ \Gamma_Q \cup (\Gamma_R \backslash x{:}B)) \subseteq \Gamma$

(iii) $(S'\ B) = D$

(iv) $(S'\ (\Delta_Q \backslash \alpha{:}A) \cup \Delta_R) \subseteq \Delta$

Proof. All similar to the proofs for the (med) case. □

By (ii) it can be seen that S' *unifies the contexts* Γ_Q *and* $(\Gamma_R \backslash x{:}B)$, *while from (iv) that it also unifies* $(\Delta_Q \backslash \alpha{:}A)$ *and* Δ_R. *It also unifies* A *and* B, *by (i) and (iii). Therefore, applying Lemma 3.3.9, we have* $\exists S$ *such that* $S' = S \circ S_U \circ S_\Delta \circ S_\Gamma$.

Now, letting $\langle \Gamma_P ; \Delta_P \rangle = \textit{pC}\,(P)$, *by the definition of the algorithm,*
$\Gamma_P = (S_U \circ S_\Delta \circ S_\Gamma\ \Gamma_Q \cup (\Gamma_R \backslash x))$ *and* $\Delta_P = (S_U \circ S_\Delta \circ S_\Gamma\ (\Delta_Q \backslash \alpha) \cup \Delta_R)$.

As in previous cases, we have:

$$\begin{aligned}(S\ \Gamma_P) &= (S{\circ}S_U{\circ}S_\Delta{\circ}S_\Gamma\ \Gamma_Q{\cup}(\Gamma_R{\setminus}x))\\ &= (S'\ \Gamma_Q{\cup}(\Gamma_R{\setminus}x))\\ &\subseteq \Gamma\end{aligned}$$

Similarly, $(S\ \Delta_P) \subseteq \Delta$*, as required.* □

A.2 Proofs for Chapter 4

Proof A.2.1 (of Proposition 4.3.5)**.** *1. Reflexivity is immediate. For transitivity, suppose that* $\forall\overrightarrow{X_i}.A{\preceq}\forall\overrightarrow{Y_j}.B$ *and* $\forall\overrightarrow{Y_j}.B{\preceq}\forall\overrightarrow{Z_k}.C$*. By definition, we have for some types* $\overrightarrow{D_i}, \overrightarrow{E_j}$ *and atomic types* $\overrightarrow{\varphi_j} \notin A$ *and* $\overrightarrow{\varphi'_k} \notin B$ *that* $B = A[\overrightarrow{D_i/X_i}]\,[\overrightarrow{Y_j/\varphi_j}]$ *and* $C = B[\overrightarrow{E_j/Y_j}]\,[\overrightarrow{Z_k/\varphi'_k}]$*. Composing these two facts, we have that* $C = A[\overrightarrow{D_i/X_i}]\,[\overrightarrow{Y_j/\varphi_j}]\,[\overrightarrow{E_j/Y_j}]\,[\overrightarrow{Z_k/\varphi'_k}]$*. Let* S *be the substitution* $\{(\overrightarrow{\varphi_j \mapsto E_j})\}$*. Then note that since* $\varphi_j \notin A$*,* $(S\ A[\overrightarrow{D_i/X_i}]) = A[\overrightarrow{(S\ D_i)/X_i}]$*. Then we have:*

$$\begin{aligned}C &= A[\overrightarrow{D_i/X_i}]\,[\overrightarrow{Y_j/\varphi_j}]\,[\overrightarrow{E_j/Y_j}]\,[\overrightarrow{Z_k/\varphi'_k}]\\ &= A[\overrightarrow{D_i/X_i}]\,[\overrightarrow{E_j/\varphi_j}]\,[\overrightarrow{Z_k/\varphi'_k}]\\ &= (S\ A[\overrightarrow{D_i/X_i}])[\overrightarrow{Z_k/\varphi'_k}]\\ &= A[\overrightarrow{(S\ D_i)/X_i}]\,[\overrightarrow{Z_k/\varphi'_k}]\end{aligned}$$

Finally, we can see that $\overrightarrow{\varphi'_k} \notin A$ *since if it were the case that* $\varphi'_k{\in}A$ *then since* $\overrightarrow{\varphi_j} \notin A$ *we would have* $\varphi'_k{\in}A[\overrightarrow{D_i/X_i}]\,[\overrightarrow{Y_j/\varphi_j}] = B$*; a contradiction.*

2. *Reflexivity and transitivity are straightforward. Anti-symmetry follows from the fact that if* $\overline{A} \lhd_{\langle\Gamma;\Delta\rangle} \overline{B}$ *and* $\overline{A}{\neq}\overline{B}$ *then* $\overline{B}$ *contains strictly more* $\forall$ *symbols than* $\overline{A}$*.*

3. *Let* $\overline{A} = \overrightarrow{X_i}.A$*. Then, since* $\overline{A}{\succeq}\overline{B}$*, we know that for some* $\overrightarrow{D_i}, \overrightarrow{Y_j}, \overrightarrow{\varphi_j}$ *we must have* $\overline{B} = \overrightarrow{\forall Y_j}.(A[\overrightarrow{D_i/X_i}]\,[\overrightarrow{Y_j/\varphi_j}])$*, with* $\overrightarrow{\varphi_j} \notin A$*. In addition, since* $\overline{B} \lhd_{\langle\Gamma;\Delta\rangle} \overline{C}$*, we must have for some* $\overrightarrow{Z_k}$ *and* $\overrightarrow{\varphi'_k} \notin \langle\Gamma;\Delta\rangle$ *that* $\overline{C} = \overrightarrow{\forall Z_k}.\overline{B}[\overrightarrow{Z_k/\varphi'_k}]$*. Since* $\overrightarrow{\varphi'_k} \notin \langle\Gamma;\Delta\rangle$ *and* $\overline{A}{\in}\langle\Gamma;\Delta\rangle$*, we have* $\overrightarrow{\varphi'_k} \notin \overline{A}$*. Hence,* $\overline{C} = \overrightarrow{\forall Z_k}.\overrightarrow{\forall Y_j}.(A[\overrightarrow{D_i/X_i}]\,[\overrightarrow{Y_j/\varphi_j}]\,[\overrightarrow{Z_k/\varphi'_k}])$ *with* $\overrightarrow{\varphi_j}, \overrightarrow{\varphi'_k} \notin \overline{A}$*, i.e.* $\overline{C}{\succeq}\overline{A}$ *as required.*

4. *Without loss of generality, say* $\overline{A} = \overrightarrow{\forall X_i}.A$ *and* $\overline{B} = \overrightarrow{\forall Y_j}.(A[\overrightarrow{C_i/X_i}]\,[\overrightarrow{Y_j/\varphi_j}])$ *(with* $\varphi_j \notin \overline{A}$*). Note that* $\varphi_j \notin \overline{B}$ *also, due to the renaming* $[\overrightarrow{Y_j/\varphi_j}]$*. Let* $S' = (S \cap \{\overrightarrow{\varphi_j}\})$*. Then we have* $(S'\ \overline{A}) = (S\ \overline{A})$ *and* $(S'\ \overline{B}) = (S\ \overline{B})$*. Therefore, it suffices to prove that* $(S'\ \overline{A}){\succeq}(S'\ \overline{B})$*. This can be seen from the fact that* $(S\ \overline{A}) = \overrightarrow{\forall X_i}.(S\ A)$ *and*

the following working:

$$\begin{aligned}
(S'\ \overline{B}) &= (S'\ \overline{\forall Y_j}.(A[\overline{C_i/X_i}]\ [\overline{Y_j/\varphi_j}])) && \textit{defn of } \overline{B}\\
&= \overline{\forall Y_j}.(S'\ (A[\overline{C_i/X_i}]\ [\overline{Y_j/\varphi_j}])) && \textit{defn of substitution}\\
&= \overline{\forall Y_j}.((S'\ A[\overline{C_i/X_i}])[\overline{Y_j/\varphi_j}]) && \textit{defn of } S'\\
&= \overline{\forall Y_j}.((S'\ A)[\overline{(S'\ C_i)/X_i}]\ [\overline{Y_j/\varphi_j}])\\
&= \overline{\forall Y_j}.((S'\ A)[\overline{D_i/X_i}]\ [\overline{Y_j/\varphi_j}]) && \textit{setting } \overline{D_i = (S'\ C_i)}
\end{aligned}$$

5. *By definition, there exist $\overline{\varphi_i}$ and $\overline{X_i}$ such that $\overline{B} = \overline{\forall X_i}.\overline{A}[\overline{X_i/\varphi_i}]$ and $\overline{\varphi_i \notin \langle \Gamma; \Delta \rangle}$. Define $S' = \{\overline{(\varphi_i \mapsto \varphi_i')}\}$, where $\overline{\varphi_i'}$ are fresh atomic types. Then we know that $\overline{B} = \overline{\forall X_i}.(S'\ \overline{A})[\overline{X_i/\varphi_i'}]$. Therefore, $(S\ \overline{B}) = (S\ (\overline{\forall X_i}.(S'\ \overline{A})[\overline{X_i/\varphi_i'}])) = \overline{\forall X_i}.(S{\circ}S'\ \overline{A})[\overline{X_i/\varphi_i'}]$. By construction, $\overline{\varphi_i' \notin \langle (S\ \Gamma); (S\ \Delta) \rangle}$, and so we conclude $(S{\circ}S'\ \overline{A}) \lhd_{\langle (S\ \Gamma);(S\ \Delta)\rangle} (S\ \overline{B})$ as required.*

6. *By definition, for some $\overline{X_i}$ and $\overline{\varphi_i} \notin \langle \Gamma; \Delta \rangle$, we must have $\overline{A} = \overline{\forall X_i}.A[\overline{X_i/\varphi_i}]$. Additionally, since $\overline{A} \succeq \overline{B}$, there must exist $\overline{C_i}$ and $\overline{\varphi_j}$ and $\overline{Y_j}$ such that we can obtain $\overline{B} = \overline{\forall Y_j}.(A[\overline{X_i/\varphi_i}])[\overline{C_i/X_i}]\ [\overline{Y_j/\varphi_j}]$ and $\overline{\varphi_j \notin \overline{A}}$. Let S be the substitution $\{\overline{(\varphi_i \mapsto C_i)}\}$. Then $\overline{B} = \overline{\forall Y_j}.(S\ A)[\overline{Y_j/\varphi_j}]$. Let $S' = \{\overline{(\varphi_j \mapsto \varphi_j')}\}$ where $\overline{\varphi_j'}$ are fresh. Then $\overline{B} = \overline{\forall Y_j}.(S'{\circ}S\ A)[\overline{Y_j/\varphi_j'}]$ and $\overline{\varphi' \notin \langle \Gamma; \Delta \rangle}$, i.e., $(S'{\circ}S\ A) \lhd_{\langle \Gamma;\Delta\rangle} \overline{B}$. Finally, since $\overline{\varphi_i} \notin \langle \Gamma; \Delta \rangle$, we have $S\ \langle \Gamma; \Delta \rangle = \langle \Gamma; \Delta \rangle$ as required.*

7. *Since $\overline{A} \succeq B$, there must exist $\overline{C_i}$ such that $B = A[\overline{C_i/X_i}]$. Let $S = \{\overline{(\varphi_i \mapsto C_i)}\}$. Then the result immediately follows.* □

Proof A.2.2 (of Proposition 4.3.7)**.** *1. By induction on the structure of the term P. We give two representative cases (all others are simpler):*

$\langle x.\alpha \rangle$**:** *Then by Lemma 4.3.8 (1), $\Gamma = \Gamma', x : \overline{A}$ and $\Delta = \alpha : \overline{B}, \Delta'$ with $\overline{A} \succeq \overline{B}$. By Proposition 4.3.5 (4), $(S\ \overline{A}) \succeq (S\ \overline{B})$. Therefore, by applying the rule (ax), we obtain $\langle x.\alpha \rangle :\cdot (S\ \Gamma'), x : (S\ \overline{A}) \vdash_{\mathrm{SP}} \alpha : (S\ \overline{B}), \Delta'$ as required.*

$\widehat{x}P\widehat{\alpha}{\cdot}\beta$**:** *Then by Lemma 4.3.8 (2), $\Delta = \beta : \overline{C}, \Delta'$ and there exist A,B such that*

$$P :\cdot \Gamma, x : A \vdash_{\mathrm{SP}} \alpha : B, \Delta' \tag{A.1}$$

and $(A{\rightarrow}B) \lhd_{\langle \Gamma;\Delta'\rangle} \overline{C}$. By Proposition 4.3.5 (5), there exists a substitution S' such that:

$$(S{\circ}S'\ (A{\rightarrow}B)) \lhd_{\langle (S\ \Gamma);(S\ \Delta')\rangle} (S\ \overline{C}) \tag{A.2}$$

$$(S'\ \langle \Gamma; \Delta \rangle) = \langle \Gamma; \Delta \rangle \tag{A.3}$$

$$(S'\ \overline{C}) = \overline{C} \tag{A.4}$$

By induction, using $(Eq.\ A.1)$ *with the substitution* $(S{\circ}S')$*, we obtain that* $P :\cdot (S{\circ}S'\ \Gamma), x : (S{\circ}S'\ A) \vdash_{\text{SP}} \alpha : (S{\circ}S'\ B), (S{\circ}S'\ \Delta')$*. Using* $(Eq.\ A.3)$ *and* $(Eq.\ A.4)$*, this gives us* $P :\cdot (S\ \Gamma), x : (S{\circ}S'\ A) \vdash_{\text{SP}} \alpha : (S{\circ}S'\ B), (S\ \Delta')$*. Furthermore, noting that* $(S{\circ}S'\ (A{\to}B)) = (S{\circ}S'\ A){\to}(S{\circ}S'\ B)$*, we can apply the rule* $(\to\mathcal{R})$ *using* $(Eq.\ A.2)$ *to obtain* $\widehat{x}P\widehat{\alpha}{\cdot}\beta :\cdot (S\ \Gamma) \vdash_{\text{SP}} \beta : (S\ \overline{C}), (S\ \Delta')$ *as required.*

2. *By induction on the structure of the term* P*. The only cases which are not straightforward are when 'closures' are taken, since we must be careful that the appropriate conditions can still be fulfilled within the larger context. This situation is exemplified by the case of a term* $\widehat{x}P\widehat{\alpha}{\cdot}\beta$*, and this is the only case we show here. As usual, by Lemma 4.3.8 (2), we obtain the statements* $P :\cdot \Gamma, x : A \vdash_{\text{SP}} \alpha : B, \Delta''$ *with* $\Delta = \beta : \overline{C}, \Delta''$ *and* $(A{\to}B) \lhd_{\langle \Gamma;\Delta''\rangle} \overline{C}$*. By unravelling the definition, we know that* $\overline{C} = \overrightarrow{\forall X_i}.(A{\to}B)[\overrightarrow{X_i/\varphi_i}]$*, for some* $\overrightarrow{X_i}$ *and some* $\overrightarrow{\varphi_i \notin \langle \Gamma; \Delta''\rangle}$*. In order to ensure that we can still 'close' the type in the larger context, we rename these atomic types: define the substitution* $S = \{\overrightarrow{(\varphi_i \mapsto \varphi_i')}\}$ *for fresh* φ_i'*. Note that* $(S\ \Gamma) = \Gamma$ *and* $(S\ \Delta'') = \Delta''$*. Then, by part 1, we obtain* $P :\cdot \Gamma, x : (S\ A) \vdash_{\text{SP}} \alpha : (S\ B), \Delta''$*. Since* x *and* α *are bound in the original term, we may assume that* $x \notin \Gamma'$ *and* $\alpha \notin \Delta'$*. Therefore,* $\langle \Gamma, x : (S\ A), \Gamma'; \alpha : (S\ B), \Delta'', \Delta'\rangle$ *is a well-formed context. By induction,* $P :\cdot \Gamma, x : (S\ A), \Gamma' \vdash_{\text{SP}} \alpha : (S\ B), \Delta'', \Delta'$*. In order to be able to apply the* $(\to\mathcal{R})$ *and conclude, it would suffice to show that* $(S\ A{\to}B) \lhd_{\langle \Gamma\cup\Gamma';\Delta\cup\Delta'\rangle} \overline{C}$*. But this follows by construction of* S*.*

3. *By straightforward induction on the structure of the derivation.*

4. *By induction on the structure of the term. We present two representative cases.*

   $P = \langle x.\alpha\rangle$**:** *By Lemma 4.3.8 (1), there exist* X*,*Δ' *such that* $\Delta = \alpha : \overline{C}, \Delta'$ *and* $\overline{B} \succeq \overline{C}$*. By Proposition 4.3.5 (1) we have* $\overline{A} \succeq \overline{C}$*. Therefore, by applying the rule* (ax)*, we obtain* $\langle x.\alpha\rangle :\cdot (\Gamma\backslash x), x : \overline{A} \vdash_{\text{SP}} \alpha : \overline{C}, \Delta$ *as required.*

   $P = \langle y.\alpha\rangle$**,** $y \neq x$**:** *This case is immediate from Lemma 4.3.8 (1).*

   $P = \widehat{y}P_1\widehat{\alpha}{\cdot}\beta$**:** *By straightforward induction, using Lemma 4.3.8 (2).*

   $P = Q\widehat{\alpha}\,[x]\,\widehat{y}R$**:** *By Lemma 4.3.8 (3) and Proposition 4.3.7 (2), there exist* C*,*D *with* $Q :\cdot \Gamma, x : \overline{B} \vdash_{\text{SP}} \alpha : \overline{C}, \Delta$ *and* $R :\cdot \Gamma, x : \overline{B}, y : D \vdash_{\text{SP}} \Delta$ *and* $\overline{B} \succeq (C{\to}D)$*. By induction,* $Q :\cdot (\Gamma\backslash x), x : \overline{A} \vdash_{\text{SP}} \alpha : \overline{C}, \Delta$ *and* $R :\cdot (\Gamma\backslash x), x : \overline{A}, y : D \vdash_{\text{SP}} \Delta$*. By Proposition 4.3.5 (1), we have* $\overline{A} \succeq (C{\to}D)$*. By the rule* $(\to\mathcal{L})$*, we obtain* $P :\cdot (\Gamma\backslash x), x : \overline{A} \vdash_{\text{SP}} \Delta$ *as required.*

5. *By induction on the structure of the term* P*. We present two representative cases.*

$P = \langle x.\alpha\rangle$**:** *By Lemma 4.3.8 (1), there exist* $\overline{C}$,Γ' *such that* $\Gamma = \Gamma', x:\overline{C}$ *and* $\overline{C} \succeq \overline{B}$. *By Proposition 4.3.5 (1),* $\overline{C} \succeq \overline{A}$. *Therefore, by the rule* (ax), *we can deduce that* $\langle x.\alpha\rangle :\cdot \Gamma', x:\overline{C} \vdash_{\text{SP}} \alpha : \overline{A}, (\Delta \backslash \alpha)$ *as required.*

$P = \widehat{x}Q\widehat{\beta}\cdot\alpha$**:** *By Lemma 4.3.8 (2), there exist* C,D *such that* $(C \rightarrow D) \lhd_{\langle\Gamma;\Delta\rangle} \overline{A}$ *and* $Q :\cdot \Gamma, x:C \vdash_{\text{SP}} \beta : D, \Delta$. *By Proposition 4.3.5 (6), there exists a substitution* S *such that* $(S\ \langle\Gamma;\Delta\rangle) = \langle\Gamma;\Delta\rangle$ *and* $(S\ C \rightarrow D) \lhd_{\langle\Gamma;\Delta\rangle} \overline{B}$. *By Proposition 4.3.7 (1) we have* $Q :\cdot \Gamma, x:(S\ C) \vdash_{\text{SP}} \beta : (S\ D), \Delta$. *We consider two cases:*

- $\alpha : \overline{B} \in \Delta$**:** *Then, by induction,* $Q :\cdot \Gamma, x:(S\ C) \vdash_{\text{SP}} \beta : (S\ D), \alpha : \overline{B}, (\Delta\backslash\alpha)$. *By the rule* $(\rightarrow\mathcal{R})$, *we obtain* $\widehat{x}Q\widehat{\beta}\cdot\alpha :\cdot \Gamma \vdash_{\text{SP}} \alpha : \overline{B}, (\Delta\backslash\alpha)$ *as required.*
- $\alpha \notin \Delta$**:** *Then, by the* $(\rightarrow\mathcal{R})$ *rule, we obtain* $\widehat{x}Q\widehat{\beta}\cdot\alpha :\cdot \Gamma \vdash_{\text{SP}} \alpha : \overline{B}, \Delta$ *as required.*

6. *By induction on the structure of the term* P, *similar to the previous part.* □

Proof A.2.3 (of Theorem 4.3.9)**.** *1. (a) By induction on the structure of the term* Q.

$Q = \langle x.\beta\rangle$**:** *Then* $Q\{P\widehat{\alpha} \leftrightarrow x\} = P\widehat{\alpha} \dagger \widehat{x}Q$ *and the result follows by application of the* (cut) *rule.*

$Q = \langle y.\beta\rangle, y \neq x$**:** *Then* $Q\{P\widehat{\alpha} \leftrightarrow x\} = Q$. *Since* $x \notin fs(Q)$, *By* $(Eq.\ 4.2)$ *and Proposition 4.3.7 (3) we obtain* $Q :\cdot \Gamma \vdash_{\text{SP}} \Delta$ *as required.*

$Q = \widehat{y}Q_1\widehat{\beta}\cdot\gamma$**:** *Then* $Q\{P\widehat{\alpha} \leftrightarrow x\} = \widehat{y}(Q_1\{P\widehat{\alpha} \leftrightarrow x\})\widehat{\beta}\cdot\gamma$. *By* $(Eq.\ 4.2)$ *and Lemma 4.3.8 (2), there exist* $B,C,\overline{D}$ *and* Δ' *such that* $\Delta = \Delta', \gamma : \overline{D}$ *and* $Q_1 :\cdot \Gamma, x:\overline{A}, y:B \vdash_{\text{SP}} \beta : C, \Delta$ *and* $(B \rightarrow C) \lhd_{\langle\Gamma, x:\overline{A};\Delta'\rangle} \overline{D}$. *From the induction hypothesis,* $Q_1\{P\widehat{\alpha} \leftrightarrow x\} :\cdot \Gamma, y:B \vdash_{\text{SP}} \beta : C, \Delta$. *Now we apply* $(\rightarrow\mathcal{R})$ *rule to obtain* $\widehat{y}(Q_1\{P\widehat{\alpha} \leftrightarrow x\})\widehat{\beta}\cdot\gamma :\cdot \Gamma \vdash_{\text{SP}} \gamma : \overline{D}, \Delta'$ *as required.*

$Q = Q_1\widehat{\beta}\ [x]\ \widehat{z}Q_2$**:**
$Q\{P\widehat{\alpha} \leftrightarrow x\} = P\widehat{\alpha} \dagger \widehat{y}((Q_1\{P\widehat{\alpha} \leftrightarrow x\})\widehat{\beta}\ [y]\ \widehat{z}(Q_2\{P\widehat{\alpha} \leftrightarrow x\}))$ *in which* y *is fresh. By* $(Eq.\ 4.2)$ *and Lemma 4.3.8 (3), there exist* $B,C,\overline{D},\Gamma'$ *such that* $\overline{D} \succeq (B \rightarrow C)$ *and* $\Gamma = \Gamma', y:\overline{D}$ *and (by applying Proposition 4.3.7 (2) as necessary)* $Q_1 :\cdot \Gamma, x:\overline{A} \vdash_{\text{SP}} \beta : B, \Delta$ *and* $Q_2 :\cdot \Gamma, x:\overline{A}, z:C \vdash_{\text{SP}} \Delta$. *By induction, twice, we obtain the judgement* $Q_1\{P\widehat{\alpha} \leftrightarrow x\} :\cdot \Gamma \vdash_{\text{SP}} \beta : B, \Delta$ *and also* $Q_2\{P\widehat{\alpha} \leftrightarrow x\} :\cdot \Gamma, z:C \vdash_{\text{SP}} \Delta$. *By* $\overline{D} \succeq (B \rightarrow C)$, *we can apply the* $(\rightarrow\mathcal{L})$ *rule to obtain* $(Q_1\{P\widehat{\alpha} \leftrightarrow x\})\widehat{\beta}\ [y]\ \widehat{z}(Q_2\{P\widehat{\alpha} \leftrightarrow x\}) :\cdot \Gamma, y:\overline{D} \vdash_{\text{SP}} \Delta$. *Finally, we apply the* (cut) *rule to obtain the required result.*

$Q = Q_1\widehat{\beta}\ [y]\ \widehat{z}Q_2, y \neq x$**:** *By straightforward induction, similar to the previous case.*

$Q = \widehat{y}Q_1 \cdot \beta$**:** *Similar to the* $\widehat{y}Q_1\widehat{\beta}\cdot\gamma$ *case.*

$Q = y \cdot Q_1\widehat{\beta}$**:** *Similar to the* $Q_1\widehat{\beta}\ [y]\ \widehat{z}Q_2$ *cases.*

$Q = Q_1\widehat{\beta}\,[x]\,\widehat{z}Q_2$**:** *Then* $Q\{P\widehat{\alpha}\leftrightarrow x\} = P\widehat{\alpha} \dagger \widehat{y}(Q_1\{P\widehat{\alpha}\leftrightarrow x\})$. *By Lemma 4.3.8 (6), there exists* $\overline{B}$ *such that both*

$$\langle x.\beta\rangle :\cdot \Gamma, x:\overline{A} \vdash_{\text{SP}} \beta:\overline{B}, \Delta \tag{A.5}$$

$$Q_1 :\cdot \Gamma, x:\overline{A}, y:\overline{B} \vdash_{\text{SP}} \Delta \tag{A.6}$$

By Lemma 4.3.8 (1), we must have

$$\overline{A} \succeq \overline{B} \tag{A.7}$$

By applying Proposition 4.3.7 (2) to $(Eq.\ A.5)$*, we obtain*

$$P :\cdot \Gamma, y:\overline{B} \vdash_{\text{SP}} \alpha:\overline{A}, \Delta \tag{A.8}$$

By applying induction to $(Eq.\ 4.2)$ *and* $(Eq.\ A.8)$*, we obtain*

$$Q_1\{P\widehat{\alpha}\leftrightarrow x\} :\cdot \Gamma, y:\overline{B} \vdash_{\text{SP}} \Delta \tag{A.9}$$

By applying Proposition 4.3.7 (4) to $(Eq.\ A.7)$ *and* $(Eq.\ A.9)$*, we obtain*

$$Q_1\{P\widehat{\alpha}\leftrightarrow x\} :\cdot \Gamma, y:\overline{A} \vdash_{\text{SP}} \Delta \tag{A.10}$$

As a final step, by applying the rule (cut) *to* $(Eq.\ 4.1)$ *and* $(Eq.\ A.10)$ *we obtain* $P\widehat{\alpha} \dagger \widehat{y}(Q_1\{P\widehat{\alpha}\leftrightarrow x\}) :\cdot \Gamma \vdash_{\text{SP}} \Delta$ *as required.*

$Q = Q_1\widehat{\beta} \dagger \widehat{y}Q_2$**,** $Q \neq \langle x.\beta\rangle$**:**
$Q\{P\widehat{\alpha}\leftrightarrow x\} = (Q_1\{P\widehat{\alpha}\leftrightarrow x\})\widehat{\beta} \dagger \widehat{y}(Q_2\{P\widehat{\alpha}\leftrightarrow x\})$.
Using $(Eq.\ 4.2)$ *and applying Lemma 4.3.8 (6), there exists a type* $\overline{B}$ *such that* $Q_1 :\cdot \Gamma, x:\alpha \vdash_{\text{SP}} \beta:\overline{B}, \Delta$ *and* $Q_2 :\cdot \Gamma, x:\overline{A}, y:\overline{B} \vdash_{\text{SP}} \Delta$*. By induction,* $Q\{P\widehat{\alpha}\leftrightarrow x\} :\cdot \Gamma \vdash_{\text{SP}} \beta:\overline{B}, \Delta$ *and* $Q_2\{P\widehat{\alpha}\leftrightarrow x\} :\cdot \Gamma \vdash_{\text{SP}} y:\overline{B}, \Delta$*. We conclude by applying the rule* (cut)*.*

(b) By induction on the structure of the term P*. The argument is similar to the previous part, and we show only the most-interesting case, where* $P = \widehat{y}P_1\widehat{\beta}\cdot\alpha$*. Then* $P\{\alpha\leftrightarrow\widehat{x}Q\} = (\widehat{y}(P_1\{\alpha\leftrightarrow\widehat{x}Q\})\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{x}Q$*, in which* γ *is fresh. By* $(Eq.\ 4.1)$ *and Lemma 4.3.8 (2), there exist* B,C *with* $P_1 :\cdot \Gamma, y:B \vdash_{\text{SP}} \beta:C, \Delta$ *and* $B{\to}C \lhd_{\langle\Gamma;\Delta\rangle} \overline{A}$*. By applying Proposition 4.3.7 (2) as necessary, we obtain* $P_1 :\cdot \Gamma, x:\overline{A}, y:B \vdash_{\text{SP}} \beta:C, \Delta$ *and* $Q :\cdot \Gamma, x:\overline{A}, y:B \vdash_{\text{SP}} \beta:C, \Delta$*. By induction,* $P_1\{\alpha\leftrightarrow\widehat{x}Q\} :\cdot \Gamma, y:B \vdash_{\text{SP}} \beta:C, \Delta$*. By the rule* $(\to\mathcal{R})$ *we obtain* $\widehat{y}(P_1\{\alpha\leftrightarrow\widehat{x}Q\})\widehat{\beta}\cdot\gamma :\cdot \Gamma \vdash_{\text{SP}} \gamma:\overline{A}, \Delta$*. Finally, by the rule* (cut) *we obtain that* $(\widehat{y}(P_1\{\alpha\leftrightarrow\widehat{x}Q\})\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{x}Q :\cdot \Gamma \vdash_{\text{SP}} \Delta$ *as required.*

2. *By inductions on the number of reduction steps, and the structure of the term P, we need only consider the case where P is the redex itself, and is reduced in one step to Q. Therefore, we show the witness reduction result for each of the reduction rules in turn:*

   (cap) : $\langle x.\alpha\rangle\widehat{\alpha} \dagger \widehat{y}\langle y.\beta\rangle \rightarrow \langle x.\beta\rangle$

   Suppose $\langle x.\alpha\rangle\widehat{\alpha} \dagger \widehat{y}\langle y.\beta\rangle :\cdot \Gamma \vdash_{\text{SP}} \Delta$. *By Lemma 4.3.8 (6),* $\alpha \notin \Delta$ *and* $x \notin \Gamma$ *and there exists* $\overline{B}$ *such that* $\langle x.\alpha\rangle :\cdot \Gamma \vdash_{\text{SP}} \alpha : \overline{B}, \Delta$ *and* $\langle y.\beta\rangle :\cdot \Gamma, y : \overline{B} \vdash_{\text{SP}} \Delta$. *By applying Lemma 4.3.8 (1) twice, there exists* $\overline{A}$,$\overline{C}$,Γ',Δ' *such that* $\Gamma = \Gamma', x : \overline{A}$ *and* $\Delta = \beta : \overline{C}, \Delta'$ *with* $\overline{A} \succeq \overline{B}$ *and* $\overline{B} \succeq \overline{C}$. *By Proposition 4.3.5 (1),* $\overline{A} \succeq \overline{C}$. *Therefore, by the rule* (ax), *we obtain* $\langle x.\beta\rangle :\cdot \Gamma', x : \overline{A} \vdash_{\text{SP}} \beta : \overline{C}, \Delta'$ *as required.*

   ($impR$) : $(\widehat{x}P\widehat{\alpha}\cdot\beta)\widehat{\beta} \dagger \widehat{y}\langle y.\gamma\rangle \rightarrow \widehat{x}P\widehat{\alpha}\cdot\gamma$ (*if* $\beta \notin fp(P)$)

   Suppose $(\widehat{x}P\widehat{\alpha}\cdot\beta)\widehat{\beta} \dagger \widehat{y}\langle y.\gamma\rangle :\cdot \Gamma \vdash_{\text{SP}} \Delta$. *By Lemma 4.3.8 (6),* $\beta \notin \Delta$ *and* $y \notin \Gamma$ *and there exists* $\overline{C}$ *such that* $\widehat{x}P\widehat{\alpha}\cdot\beta :\cdot \Gamma \vdash_{\text{SP}} \beta : \overline{C}, \Delta$ *and* $\langle y.\gamma\rangle :\cdot \Gamma, y : \overline{C} \vdash_{\text{SP}} \Delta$. *By Lemma 4.3.8 (2), there exist* A,B,Δ'' *such that* $(A \rightarrow B) \lhd_{\langle\Gamma;\Delta\rangle} \overline{C}$ *and also* $P :\cdot \Gamma, x : A \vdash_{\text{SP}} \alpha : B, \Delta''$ *and* $(\beta : \overline{C}, \Delta) = (\beta : \overline{C}, \Delta'')$. *Since* $\beta \notin P$, *using Proposition 4.3.7 (3), we may assume without loss of generality that we have* $\beta \notin \Delta''$, *and therefore that* $\Delta'' = \Delta$. *By Lemma 4.3.8 (1), there exist* $\overline{D}$,Δ' *such that* $\Delta = \gamma : \overline{D}, \Delta'$ *and* $\overline{C} \succeq \overline{D}$. *By Proposition 4.3.5 (6), there exists a substitution* S *such that* $(S\ \langle\Gamma;\Delta\rangle) = \langle\Gamma;\Delta\rangle$ *and* $(S\ A \rightarrow B) \lhd_{\langle\Gamma;\Delta\rangle} \overline{D}$. *By Proposition 4.3.7 (1), we have* $P :\cdot \Gamma, x : (S\ A) \vdash_{\text{SP}} \alpha : (S\ B), \gamma : \overline{D}, \Delta'$. *By the rule* ($\rightarrow\mathcal{R}$), *we deduce that* $\widehat{x}P\widehat{\alpha}\cdot\gamma :\cdot \Gamma \vdash_{\text{SP}} \gamma : \overline{D}, \Delta$ *as required.*

   ($impL$) : $\langle x.\alpha\rangle\widehat{\alpha} \dagger \widehat{y}(P\widehat{\beta}\,[y]\,\widehat{z}Q) \rightarrow P\widehat{\beta}\,[x]\,\widehat{z}Q$ (*if* $y \notin fs(P, Q)$)

   Suppose $\langle x.\alpha\rangle\widehat{\alpha} \dagger \widehat{y}(P\widehat{\beta}\,[y]\,\widehat{z}Q) :\cdot \Gamma \vdash_{\text{SP}} \Delta$. *By Lemma 4.3.8 (6),* $\alpha \notin \Delta$ *and* $y \notin \Gamma$ *and there exists* $\overline{B}$ *such that we have both* $\langle x.\alpha\rangle :\cdot \Gamma \vdash_{\text{SP}} \alpha : \overline{B}, \Delta$ *and* $P\widehat{\beta}\,[y]\,\widehat{z}Q :\cdot \Gamma, y : \overline{B} \vdash_{\text{SP}} \Delta$. *By Lemma 4.3.8 (1), there exist* $\overline{A}$,Γ' *such that* $\Gamma = \Gamma', x : \overline{A}$ *and* $\overline{A} \succeq \overline{B}$. *By Lemma 4.3.8 (3), there exist* C,D,Γ'' *such that* $(\Gamma, y : \overline{B}) = (\Gamma'', y : \overline{B})$ *and* $\overline{B} \succeq (C \rightarrow D)$ *and* $P :\cdot \Gamma'' \vdash_{\text{SP}} \beta : C, \Delta$ *and also* $Q :\cdot \Gamma'', z : D \vdash_{\text{SP}} \Delta$. *By Proposition 4.3.7 (3) we can assume without loss of generality that* $\Gamma'' = \Gamma$. *Since* $\overline{A} \succeq \overline{B} \succeq (C \rightarrow D)$, *by Proposition 4.3.5 (1) we can deduce* $\overline{A} \succeq (C \rightarrow D)$. *By applying the rule* ($\rightarrow\mathcal{L}$) *we can finally obtain* $P\widehat{\beta}\,[x]\,\widehat{z}Q :\cdot \Gamma', x : \overline{A} \vdash_{\text{SP}} \Delta$ *as required.*

   ($not\text{-}right$) : $(\widehat{x}P \cdot \alpha)\widehat{\alpha} \dagger \widehat{y}\langle y.\beta\rangle \rightarrow \widehat{x}P \cdot \beta$ (*if* $\alpha \notin fp(P)$)

   Similar to the ($impR$) *case above.*

   ($not\text{-}left$) : $\langle x.\alpha\rangle\widehat{\alpha} \dagger \widehat{y}(y \cdot P\widehat{\beta}) \rightarrow x \cdot P\widehat{\beta}$ (*if* $y \notin fs(P)$)

   Similar to the ($impL$) *case above.*

(imp) : $(\widehat{x}P\widehat{\alpha}\cdot\beta)\widehat{\beta}\dagger\widehat{y}(Q\widehat{\gamma}\,[y]\,\widehat{z}R) \rightarrow \left\{\begin{array}{l}(Q\widehat{\gamma}\dagger\widehat{x}P)\widehat{\alpha}\dagger\widehat{z}R\\ Q\widehat{\gamma}\dagger\widehat{x}(P\widehat{\alpha}\dagger\widehat{z}R)\end{array}\right\}$ ($\mathit{if}\ \beta\notin\mathit{fp}(P)$, $y\notin\mathit{fs}(Q,R)$)

Suppose $(\widehat{x}P\widehat{\alpha}\cdot\beta)\widehat{\beta}\dagger\widehat{y}(Q\widehat{\gamma}\,[y]\,\widehat{z}R) :\cdot \Gamma\vdash_{\mathrm{SP}}\Delta$. *By Lemma 4.3.8 (6),* $\beta\notin\Delta$ *and* $y\notin\Gamma$ *and there exists* $\overline{C}$ *such that we have both* $\widehat{x}P\widehat{\alpha}\cdot\beta :\cdot \Gamma\vdash_{\mathrm{SP}}\beta:\overline{C},\Delta$ *and* $Q\widehat{\gamma}\,[y]\,\widehat{z}R :\cdot \Gamma, y:\overline{C}\vdash_{\mathrm{SP}}\Delta$. *By Lemma 4.3.8 (2), there exist A,B,*Δ' *such that* $A{\rightarrow}B \lhd_{\langle\Gamma;\Delta\rangle} \overline{C}$ *and* $P :\cdot \Gamma, x:A\vdash_{\mathrm{SP}}\alpha:B,\Delta'$ *and* $(\beta:\overline{C},\Delta) = (\beta:\overline{C},\Delta')$. *As in previous cases, w.l.o.g.* $\Delta' = \Delta$ *since* $\beta\notin\Gamma$ *and* $\beta\notin\mathit{fs}(P)$. *Now, by Lemma 4.3.8 (3) and similar argument, there exist D,E such that* $\overline{C}\succeq(D{\rightarrow}E)$ *and* $Q :\cdot \Gamma\vdash_{\mathrm{SP}}\gamma:D,\Delta$ *and* $R :\cdot \Gamma, z:E\vdash_{\mathrm{SP}}\Delta$. *By Proposition 4.3.5 (6), there exists a substitution* S *such that* $(S\ \langle\Gamma;\Delta\rangle) = \langle\Gamma;\Delta\rangle$ *and* $(S\ A{\rightarrow}B) \lhd_{\langle\Gamma;\Delta\rangle} (D{\rightarrow}E)$. *In particular,* $(S\ A) = D$ *and* $(S\ B) = E$. *By Proposition 4.3.7 (1), we obtain* $P :\cdot \Gamma, x:D\vdash_{\mathrm{SP}}\alpha:E,\Delta$. *Now, by applying Proposition 4.3.7 (2) and the rule* (cut) *repeatedly, we first obtain both* $Q\widehat{\gamma}\dagger\widehat{x}P :\cdot \Gamma\vdash_{\mathrm{SP}}\alpha:E,\Delta$ *and* $P\widehat{\alpha}\dagger\widehat{z}R :\cdot \Gamma, x:D\vdash_{\mathrm{SP}}\Delta$, *and then both* $(Q\widehat{\gamma}\dagger\widehat{x}P)\widehat{\alpha}\dagger\widehat{z}R :\cdot \Gamma\vdash_{\mathrm{SP}}\Delta$ *and* $Q\widehat{\gamma}\dagger\widehat{x}(P\widehat{\alpha}\dagger\widehat{z}R) :\cdot \Gamma\vdash_{\mathrm{SP}}\Delta$ *as required.*

(not) : $(\widehat{x}P\cdot\alpha)\widehat{\alpha}\dagger\widehat{y}(y\cdot Q\widehat{\beta}) \rightarrow Q\widehat{\beta}\dagger\widehat{x}P$ ($\mathit{if}\ \alpha\notin\mathit{fp}(P), y\notin\mathit{fs}(Q)$)

Similar to the (imp) *case above.*

(prop-R) : $P\widehat{\alpha}\dagger\widehat{x}Q \rightarrow Q\{P\widehat{\alpha}{\leftarrow\!\!\rightarrow}x\}$ ($\mathit{if}\ Q$ *does not introduce* x)

By Lemma 4.3.8 (6) and part 1a.

(prop-L) : $P\widehat{\alpha}\dagger\widehat{x}Q \rightarrow P\{\alpha{\leftarrow\!\!\rightarrow}\widehat{x}Q\}$ ($\mathit{if}\ P$ *does not introduce* α)

By Lemma 4.3.8 (6) and part 1b. □

Proof A.2.4 (of Proposition 4.3.16)**.**

1. *Immediate from the definition, since* $\overline{B}[\overrightarrow{\varphi_i/X_i}] = \overline{A}$.

2. *Let* $\overline{C} = \overrightarrow{\forall Y_i.}\,\overline{A}[\overrightarrow{Y_i/\varphi_i}]$. *Let* $\{\overrightarrow{\varphi_j}\}$ *be the subset of* $\{\overrightarrow{\varphi_i}\}$ *which actually occur in* $\overline{A}$. *Without loss of generality, replace all of the other atomic types in* $\{\overrightarrow{\varphi_i}\}$ *with fresh atomic types. By the definition of closure, we have* $\overrightarrow{\varphi_i\notin\langle\Gamma;\Delta\rangle}$. *Now let* $\{\overrightarrow{\varphi_k}\}$ *be the set of atomic types occurring in* $\overline{A}$ *but not in* $\langle\Gamma;\Delta\rangle$. *Then* $\{\overrightarrow{\varphi_j}\}\subseteq\{\overrightarrow{\varphi_k}\}$, *and for some* $\{\overrightarrow{X_k}\}$, $\overline{B} = \overrightarrow{\forall X_k.}.\overline{A}[\overrightarrow{X_k/\varphi_k}]$. *Then* $\overline{C} = \overrightarrow{\forall Y_i.}\overline{B}[\overrightarrow{\varphi_k/X_k}]\,[\overrightarrow{Y_i/\varphi_i}]$ *as required.*

3. *Write* $\overline{B} = \overrightarrow{\forall X_i.}\,\overline{A}[\overrightarrow{X_i/\varphi_i}]$, *where* $\{\overrightarrow{\varphi}\}$ *are the atomic types occurring in* $\overline{A}$ *but not in* $\langle\Gamma;\Delta\rangle$. *Now let* $\overline{C} = \forall$*-closure* $(S\ \overline{A})\ \langle(S\ \Gamma);(S\ \Delta)\rangle = \overrightarrow{\forall Y_j.}(S\ \overline{A})[\overrightarrow{Y_j/\varphi_j}]$, *where* $\{\varphi_j\}$ *are the atomic types occurring in* $\overline{A}$ *but not in* $\langle(S\ \Gamma);(S\ \Delta)\rangle$. *Then we aim to show* $(S\ \overline{B})\succeq\overline{C}$. *This follows because* $(S\ \overline{A}) = (S\ \overline{B})[\overrightarrow{((S\ \varphi_i)/X_i}]$ *and so we have* $\overline{C} = \overrightarrow{\forall Y_j.}(S\ \overline{B})[\overrightarrow{((S\ \varphi_i)/X_i}]\ \overrightarrow{[Y_j/\varphi_j}]$. *Finally, we must be sure that* $\overrightarrow{\varphi_j}\notin(S\ \overline{B})$. *Suppose that there is some* $\varphi_j\in(S\ \overline{B})$ *(and we will show a contradiction). Then, by Lemma 4.3.18 (3), there are two possible cases:*

$\varphi_j \in \overline{B}$ **and** $(S\ \varphi_j) = \varphi_j$**:** *Then, since* $\overline{B} = \forall$*-closure* $\overline{A}\ \langle \Gamma; \Delta \rangle$*, we must have* $\varphi_j \in \langle \Gamma; \Delta \rangle$*. However, then* $\varphi_j \in \langle (S\ \Gamma); (S\ \Delta) \rangle$*, contradicting the definition of* $\overline{C}$*.*

$\exists \varphi \in \overline{B}$ **with** $\varphi_j \in (S\ \varphi)$**:** *Then, since* $\overline{B} = \forall$*-closure* $\overline{A}\ \langle \Gamma; \Delta \rangle$*, we must have* $\varphi \in \langle \Gamma; \Delta \rangle$*. But then* $\varphi_j \in \langle (S\ \Gamma); (S\ \Delta) \rangle$*, contradicting the definition of* $\overline{C}$*.*

4. *Follows easily from the observation that for any atomic type* φ *and types* $\overline{A} \succeq \overline{B}$*,* $\varphi \in \overline{A} \Rightarrow \varphi \in \overline{B}$*.* □

Proof A.2.5 (of Theorem 4.3.21)**.** *1. Let* $\overline{A} = \overrightarrow{\forall Y_j}.A$ *and* $\overline{B} = \overrightarrow{\forall Z_k}.B$*. In accordance with the definition of the algorithm, let* $A' = \mathit{freshInst}(\overline{A}) = A[\overrightarrow{\varphi_j/Y_j}]$ *and* $B' = \mathit{freshInst}(\overline{B}) = B[\overrightarrow{\varphi_k/Z_k}]$*. Since the call succeeds, we must have that the call unify* $A'\ B'$ *succeeds, yielding a substitution*

$$S_u = \mathit{unify}\ A'\ B' \tag{A.11}$$

Let $C_u = (S_u\ A')$*. Note that by soundness of unification (Lemma 3.3.9) we have* $(S_u\ B') = (S_u\ A') = C_u$*.*

Define a set of atomic types $\Psi = \{\overrightarrow{\varphi_i}\} = \mathit{atoms}(C_u) \backslash (\mathit{atoms}(S_u\ \overline{A}) \cup \mathit{atoms}(S_u\ \overline{B}))$*. We have that* $\overline{C} = \overrightarrow{\forall Z_i}.C_u[\overrightarrow{Z_i/\varphi_i}]$ *while* $S = (S_u \cap (\mathit{atoms}(\overline{A}) \cup \mathit{atoms}(\overline{B})))$*.*

We will now show that $(S\ \overline{A}) \succeq \overline{C}$*. The argument that also* $(S\ \overline{B}) \succeq \overline{C}$ *is analogous and will be omitted.*

Notice firstly that (using Lemma 4.3.19 (1)), we have $(S_r\ \overline{A}) = (S_u\ \overline{A}) = \overrightarrow{\forall Y_j}.(S_u\ A)$*. Define a set of types* $\overrightarrow{D_j = (S_u\ \varphi_j)}$ *. Then by construction, we have that:*

$$C_u = (S_u\ A[\overrightarrow{\varphi_j/Y_j}]) = (S_u\ A)[\overrightarrow{(S_u\ \varphi_j)/Y_j}] = (S_u\ A)[\overrightarrow{D_j/Y_j}]$$

Therefore, $C_u[\overrightarrow{Z_i/\varphi_i}] = (S_u\ A)\overrightarrow{[D_j/Y_j]}\ \overrightarrow{[Z_i/\varphi_i]}$*. Furthermore, by the definition of the set* Ψ*, we have* $\overrightarrow{\varphi_i \notin (S_u\ \overline{A})}$ *. Therefore, by Definition 4.2.5,* $\overrightarrow{\forall Y_j}.(S_u\ A) \succeq \overline{C}$*. Since we know* $(S_r\ \overline{A}) = \overrightarrow{\forall Y_j}.(S_u\ A)$*, we have* $(S_r\ \overline{A}) \succeq \overline{C}$ *as required.*

2. *Firstly, let us define (in which all of the* $\overrightarrow{\varphi_j}, \overrightarrow{\varphi_k}, \overrightarrow{\varphi_l}$ *are fresh):*

$$\overline{A} = \overrightarrow{\forall Y_j}.A \tag{A.12}$$

$$\overline{B} = \overrightarrow{\forall Z_k}.B \tag{A.13}$$

$$\overline{D} = \overrightarrow{\forall W_l}.D \tag{A.14}$$

$$A' = \mathit{freshInst}(\overline{A}) = A[\overrightarrow{\varphi_j/Y_j}] \tag{A.15}$$

$$B' = \mathit{freshInst}(\overline{B}) = B[\overrightarrow{\varphi_k/Z_k}] \tag{A.16}$$

$$D' = \mathit{freshInst}(\overline{D}) = D[\overrightarrow{\varphi_l/W_l}] \tag{A.17}$$

Since $(S\ \overline{A}) \succeq \overline{D}$*, we know (from Definition 4.2.5) that, for some* $\overrightarrow{E'_j}$ *and some* $\overrightarrow{\varphi'_l} \notin$ *atoms*$(S\ \overline{A})$*, we have* $D = (S\ A)[\overrightarrow{E'_j/Y_j}]\,[\overrightarrow{W_l/\varphi'_l}]$*. Define the substitution* $S_A = \{(\overrightarrow{\varphi'_l \mapsto \varphi_l})\}$*, and define the types* $\overrightarrow{E_j = (S_A\ E'_j)}$*. Then we obtain that* $D = (S\ A)[\overrightarrow{E_j/Y_j}]\,[\overrightarrow{W_l/\varphi_l}]$*. Notice that*

$$\overrightarrow{\varphi_l \notin \textit{atoms}(S\ \overline{A})} \tag{A.18}$$

since the $\overrightarrow{\varphi_l}$ *were chosen to be fresh.*

In a similar fashion, from the fact that $(S\ \overline{B}) \succeq \overline{D}$ *we can deduce that, for some types* $\overrightarrow{F_k}$ *we have* $D = (S\ B)[\overrightarrow{F_k/Z_k}]\,[\overrightarrow{W_l/\varphi_l}]$*, and that*

$$\overrightarrow{\varphi_l \notin \textit{atoms}(S\ \overline{B})} \tag{A.19}$$

Since $D' = D[\overrightarrow{\varphi_l/W_l}]$*, we deduce from the above that* $(S\ A)[\overrightarrow{E_j/Y_j}] = D' = (S\ B)[\overrightarrow{F_k/Z_k}]$*. Define next the two substitutions*

$$S_E = \{(\overrightarrow{\varphi_j \mapsto E_j})\} \tag{A.20}$$

$$S_F = \{(\overrightarrow{\varphi_k \mapsto E_k})\} \tag{A.21}$$

By construction, we have $(S_F \circ S_E \circ S\ A') = (S\ A)[\overrightarrow{E_j/Y_j}] = D' = (S_F \circ S_E \circ S\ B')$*. Therefore, the substitution* $(S_F \circ S_E \circ S)$ *is a unifier for the types* A' *and* B'*. By completeness of unification (Lemma 3.3.9), there exist substitutions* S_1 *and* S_u *such that*

$$(S_F \circ S_E \circ S) = (S_1 \circ S_u) \tag{A.22}$$

$$S_u = \textit{unify}\ A'\ B' \tag{A.23}$$

In particular, the call unify $A'\ B'$ *does not fail, and so neither does unifyGen* $\overline{A}\ \overline{B}$ *in question. Therefore, there exist* $(S_r, \overline{C}) =$ *unifyGen* $\overline{A}\ \overline{B}$*, where:*

$$S_r = (S_u \cap (\textit{atoms}(\overline{A}) \cup \textit{atoms}(\overline{B}))) \tag{A.24}$$

$$\{\overrightarrow{\varphi_i}\} = \textit{atoms}(S_u\ A') \backslash (\textit{atoms}(S_u\ \overline{A}) \cup \textit{atoms}(S_u\ \overline{B})) \tag{A.25}$$

$$\overline{C} = \overrightarrow{\forall X_i}.(S_u\ A')[\overrightarrow{X_i/\varphi_i}] \tag{A.26}$$

For convenience, we define $C' = (S_u\ A')$*, so that* $\overline{C} = \overrightarrow{\forall X_i}.C'[\overrightarrow{X_i/\varphi_i}]$*.*

We seek next to show that $(S_1\ \overline{C}) \succeq \overline{D}$*, from which we will be able to obtain the desired result without too much trouble. We would like to begin by showing that* $(S_1\ \overline{C}) = \overrightarrow{\forall X_i}.(S_1 \circ S_u\ A')[\overrightarrow{X_i/\varphi_i}]$*. However, this is not necessarily true, since we have no guarantee that the* φ_i*s are not affected by the substitution* S_1*. We choose*

to work around this, by choosing a new set $\overrightarrow{\varphi'_i}$ of fresh atomic types (one for each atomic type φ_i) and employing a renaming substitution

$$S_2 = \{\overrightarrow{(\varphi_i \mapsto \varphi'_i)}\} \tag{A.27}$$

We can now see instead that

$$(S_1\ \overline{C}) = \overrightarrow{\forall X_i}.(S_1 \circ S_2 \circ S_u\ A')\overrightarrow{[X_i/\varphi'_i]} \tag{A.28}$$

To be able to deduce that $(S_1\ \overline{C}) \succeq \overline{D}$, then (by Definition 4.2.5), we require a set of types $\overrightarrow{G_i}$ such that $D = (((S_1 \circ S_2 \circ S_u\ A')\overrightarrow{[X_i/\varphi'_i]})\overrightarrow{[G_i/X_i]}\overrightarrow{[W_l/\varphi_l]}$, and to show also that $\overrightarrow{\varphi_l \notin (S_1\ \overline{C})}$.

We claim that if we define the types $\overrightarrow{G_i = (S_1\ \varphi_i)}$ then these will do the trick. Firstly, if we define the substitution $S_G = \{\overrightarrow{(\varphi'_i \mapsto S_1\ \varphi_i)}\}$ then we can show that $(((S_1 \circ S_2 \circ S_u\ A')\overrightarrow{[X_i/\varphi'_i]})\overrightarrow{[G_i/X_i]}\overrightarrow{[W_l/\varphi_l]} = D$ as follows:

$$\begin{array}{ll}
(((S_1 \circ S_2 \circ S_u)\ A')\overrightarrow{[X_i/\varphi'_i]})\overrightarrow{[(S_1\ \varphi_i)/X_i]}\ \overrightarrow{[W_l/\varphi_l]} & \\
= ((S_G \circ S_1 \circ S_2 \circ S_u)\ A')\overrightarrow{[W_l/\varphi_l]} & \textit{compose } \overrightarrow{[X_i/\varphi'_i]}, \overrightarrow{[(S_1\ \varphi_i)/X_i]} \\
= ((S_1 \circ S_G \circ S_2 \circ S_u)\ A')\overrightarrow{[W_l/\varphi_l]} & \textit{Lemma 4.3.18 (1)} \\
= ((S_1 \circ (S_1 \cap \{\overrightarrow{\varphi_i}\}) \circ S_u)\ A')\overrightarrow{[W_l/\varphi_l]} & \textit{Lemma 4.3.19 (6)} \\
= (((S_1 \backslash \{\overrightarrow{\varphi_i}\}) \circ (S_1 \cap \{\overrightarrow{\varphi_i}\}) \circ S_u)\ A')\overrightarrow{[W_l/\varphi_l]} & \textit{idempotency, Lemma 4.3.19 (2)} \\
= ((S_1 \circ S_u)\ A')\overrightarrow{[W_l/\varphi_l]} & \textit{Lemma 4.3.19 (7)} \\
= ((S_F \circ S_E \circ S\ A')\overrightarrow{[W_l/\varphi_l]} & (S_F \circ S_E \circ S = S_1 \circ S_u) \\
= D'\overrightarrow{[W_l/\varphi_l]} & (D' = S_F \circ S_E \circ S\ A') \\
= D &
\end{array}$$

We need to also show that $\overrightarrow{\varphi_l \notin (S_1\ \overline{C})}$, which, combined with the argument above would justify that $(S_1\ \overline{C}) \succeq \overline{D}$. We will argue by contradiction; assuming that for some $\varphi_l \in \{\overrightarrow{\varphi_l}\}$ we have

$$\varphi_l \in (S_1\ \overline{C}) \tag{A.29}$$

we will show that a contradiction inevitably follows. By $(Eq.\ A.29)$ and $(Eq.\ A.28)$, we deduce that

$$\varphi_l \in \overrightarrow{\forall X_i}.(S_1 \circ S_2 \circ S_u\ A')\overrightarrow{[X_i/\varphi'_i]} \tag{A.30}$$

Then, by Lemma 4.3.18 (4), we must have 4.3.18 (3), we identify two cases:

Case 1: $\varphi_l \notin$ ***dom***(S_1) **and** $\varphi_l \in (S_2 \circ S_u\ A')$**:** *Then since $(Eq.\ A.27)$ $\varphi_l \notin \{\overrightarrow{\varphi'_i}\}$, by*

Lemma 4.3.18 (3) again, we must have

$$\varphi_l \in \textit{atoms}(S_u\ A') \tag{A.31}$$

But from the freshness of φ_l at $(Eq.\ A.17)$ we must have: $\varphi_l \notin \textit{atoms}(A')$ and $\varphi_l \notin \textit{atoms}(B')$. Furthermore, by $(Eq.\ A.23)$, we can assume also that $\varphi_l \notin \textit{atoms}(S_u\ A')$, contradicting $(Eq.\ A.31)$

Case 2: $\exists \varphi, H$ **s.t.** $\varphi \in$ ***atoms***$(S_2 \circ S_u\ A')$ **and** $(\varphi \mapsto H) \in S_1$ **and** $\varphi_l \in$ ***atoms***(H)
We must have $\varphi \notin \{\overrightarrow{\varphi_i}\}$ since this set of atomic types was chosen to be fresh at $(Eq.\ A.27)$. Therefore, by Lemma 4.3.18 (3), we must have $\varphi \in (S_u\ A')$ and $\varphi \notin \{\overrightarrow{\varphi_i}\}$. By $(Eq.\ A.25)$ it must be the case that either $\varphi \in (S_u\ \overline{A})$ or $\varphi \in (S_u\ \overline{B})$. But then, by the assumptions of this case, either $\varphi_l \in (S_1 \circ S_u\ \overline{A})$ or $\varphi_l \in (S_1 \circ S_u\ \overline{B})$. By $(Eq.\ A.22)$, $(Eq.\ A.20)$ and $(Eq.\ A.21)$, we have that either $\varphi_l \in (S\ A)$ or $\varphi_l \in (S\ B)$, contradicting $(Eq.\ A.18)$ and $(Eq.\ A.19)$, respectively.

This completes the argument justifying that

$$(S_1\ \overline{C}) \succeq \overline{D} \tag{A.32}$$

We now need to work on the form of the substitutions involved. We will employ the set of atomic types $\Psi = \{\overrightarrow{\varphi_j}\} \cup \{\overrightarrow{\varphi_k}\}$, noting that, by $(Eq.\ A.20)$ and $(Eq.\ A.21)$, we know

$$\Psi = \textit{dom}(S_F \circ S_E) \tag{A.33}$$

We can then show:

$$\begin{aligned}
S_F \circ S_E \circ S &= S_1 \circ S_u \qquad (Eq.\ A.22) \\
\therefore \quad (S_F \circ S_E \circ S) \backslash \Psi &= (S_1 \circ S_u) \backslash \Psi \\
\therefore \ ((S_F \circ S_E) \backslash \Psi) \circ S &= (S_1 \backslash \Psi) \circ S_u \ \textit{Lemma 4.3.19 (4)} \\
\therefore \qquad \textit{id} \circ S &= (S_1 \backslash \Psi) \circ S_u \ \textit{Lemma 4.3.19 (3)}
\end{aligned}$$

If we define the substitution

$$S_3 = (S_1 \backslash \Psi) \tag{A.34}$$

then we have

$$S = S_3 \circ S_u \tag{A.35}$$

Furthermore, since $\Psi \cap \textit{atoms}(\overline{C}) = \emptyset$, by $(Eq.\ A.32)$, $(Eq.\ A.34)$ and Lemma 4.3.19 (2), we obtain

$$(S_3\ \overline{C}) \succeq \overline{D} \tag{A.36}$$

We are almost done, but the substitution actually returned from the call is $S_r = (S_u \cap (\mathit{atoms}(\overline{A}) \cup \mathit{atoms}(\overline{B})))$ *as defined in* $(Eq.\ A.24)$.

We now write $\Phi = (\mathit{atoms}(\overline{A}) \cup \mathit{atoms}(\overline{B}))$ *and deduce by Lemma 4.3.19 (7) that*

$$S_3 \circ S_u = S_3 \circ (S_u \cap \Phi) \circ (S_u \backslash \Phi) \tag{A.37}$$

Finally, define $S_4 = S_3 \circ (S_u \cap \Phi)$. *We observe that:*

$$\begin{aligned} (S_4\ \overline{C}) &= (S_3 \circ (S_u \cap \Phi)\ \overline{C}) \\ &= (S_3\ \overline{C}) && \textit{Lemma 4.3.19 (5)} \\ &\succeq \overline{D} && (Eq.\ A.36) \end{aligned}$$

Therefore, we have that $S = S_4 \circ S_r$ *and* $(S_4\ \overline{C}) \succeq \overline{D}$, *as required.* □

Proof A.2.6 (of Theorem 4.3.24)**.** *1. By induction on the structure of the term* R.

$R = \langle x.\alpha \rangle$**:** *Let* $\overline{A} =$ *typeof* $x\ \Gamma$. *From the definition of the algorithm, we have* $S_R = id$ *and* $\Delta_R = \{\alpha : \overline{A}\}$. *By* $(Eq.\ 4.3)$, *we have* $\Gamma = \Gamma, x : \overline{A}$. *Then, by the rule* (ax), *we have* $\langle x.\alpha \rangle :\cdot \Gamma, x : \overline{A} \vdash_{\text{SP}} \alpha : \overline{A}$ *as required.*

$R = \widehat{x} P \widehat{\alpha} \cdot \beta$**:** *In accordance with the algorithm, let:*

$$\varphi = \mathit{fresh} \tag{A.38}$$

$$\langle S_P, \Delta_P \rangle = \text{sppc}\ (P, \Gamma \cup \{x : \varphi\}) \tag{A.39}$$

$$A = (S_P\ \varphi) \tag{A.40}$$

$$B = \mathit{freshInstance\ typeof}\ \alpha\ \Delta_P \tag{A.41}$$

$$\overline{C} = \forall\text{-}\mathit{closure}\ A \rightarrow B\ \langle (S_P\ \Gamma); \Delta_P \backslash \alpha \rangle \tag{A.42}$$

$$\langle S_u, \overline{D} \rangle = \begin{cases} \mathit{unifyGen}\ \overline{C}\ \mathit{typeof}\ \beta\ \Delta_P & \mathit{if}\ \beta \in \Delta_P \\ \langle \mathit{id}, \overline{C} \rangle \mathit{otherwise} \end{cases} \tag{A.43}$$

$$S_r = (S_u \circ S_P \cap \mathit{atoms}(\Gamma)) \tag{A.44}$$

$$\text{sppc}\ (\widehat{x} P \widehat{\alpha} \cdot \beta, \Gamma) = \langle S_r, (S_u\ \Delta_P \backslash \alpha \backslash \beta) \cup \{\beta : \overline{D}\} \rangle \tag{A.45}$$

By induction, using $(Eq.\ A.39)$, *we have*

$$P :\cdot (S_P\ \Gamma \cup \{x : \varphi\}) \vdash_{\text{SP}} \Delta_P \tag{A.46}$$

By applying Propositions 4.3.7 (2) and 5 to $(Eq.\ A.46)$ *as appropriate, and using* $(Eq.\ A.40)$ *and* $(Eq.\ A.41)$, *we obtain*

$$P :\cdot (S_P\ \Gamma), x : A \vdash_{\text{SP}} (\Delta_P \backslash \alpha), \alpha : B \tag{A.47}$$

.

We wish now to apply the type-assignment rule (*impR*). *However, examining the conclusion of this rule, we need to ensure that the resulting right-context will be well-formed, i.e. deal with the possibility that* $\beta \in \Delta_P$ *already. To do this, we consider two cases:*

$\beta \in \Delta_P$**:** *Then, by* $(Eq.\ A.43)$ *we have*

$$\langle S_u, \overline{D} \rangle = \textit{unifyGen}\ \overline{C}\ \textit{typeof}\ \beta\ \Delta_P \tag{A.48}$$

Since the original call to $\text{sppc}\,(R, \Gamma)$ *was assumed to succeed, this sub-call to unifyGen must also succeed, so such a pair exists. By the soundness of unifyGen (Theorem 4.3.21 (1)), we have that*

$$(S_u\ \overline{C}) \succeq \overline{D} \tag{A.49}$$

$$(S_u\ \textit{typeof}\ \beta\ \Delta_P) \succeq \overline{D} \tag{A.50}$$

By Proposition 4.3.7 (1), and $(Eq.\ A.47)$*, we have*

$$P :\cdot (S_u\ S_P\ \Gamma), x : (S_u\ A) \vdash_{\text{SP}} (S_u\ (\Delta_P \backslash \alpha)), \alpha : (S_u\ B) \tag{A.51}$$

By Proposition 4.3.7 (4), we obtain
$P :\cdot (S_u \circ S_P\ \Gamma), x : (S_u\ A) \vdash_{\text{SP}} (S_u\ (\Delta_P \backslash \alpha \backslash \beta)), \beta : \overline{D}, \alpha : (S_u\ B)$ *and also* $(S_u\ A \rightarrow B) \lhd_{\langle (S_u \circ S_P\ \Gamma); (S_u\ (\Delta_P \backslash \alpha \backslash \beta)), \beta : \overline{D} \rangle}$. *Therefore, by applying the rule* $(\rightarrow\mathcal{R})$*, we obtain* $\widehat{x} P \widehat{\alpha} \cdot \beta :\cdot (S_u \circ S_p\ \Gamma) \vdash_{\text{SP}} (S_u\ \Delta_P \backslash \alpha \backslash \beta), \beta : \overline{D}$.

$\beta \notin \Delta_P$**:** *Then, by* $(Eq.\ A.43)$*, we have* $S_u = \textit{id}$ *and* $\overline{D} = \overline{C}$*. Furthermore, since* $\beta \notin \Delta_P$*, by applying Proposition 4.3.16 to* $(Eq.\ A.42)$*, and applying the rule* (*impR*) *to* $(Eq.\ A.47)$*, we obtain*

$$\widehat{x} P \widehat{\alpha} \cdot \beta :\cdot (S_P\ \Gamma) \vdash_{\text{SP}} (\Delta_P \backslash \alpha), \beta : \overline{D} \tag{A.52}$$

Therefore, trivially we have $\widehat{x} P \widehat{\alpha} \cdot \beta :\cdot (S_u \circ S_p\ \Gamma) \vdash_{\text{SP}} (S_u\ \Delta_P \backslash \alpha \backslash \beta), \beta : \overline{D}$.

We conclude the case, noting that $(S_r\ \Gamma) = (S_u \circ S_p\ \Gamma)$ *by definition of* S_r.

$R = P\widehat{\alpha}\,[x]\,\widehat{y}Q$**:** *In accordance with the algorithm, let:*

$$\langle S_P, \Delta_P\rangle = \text{sppc}\,(P, \Gamma) \tag{A.53}$$

$$\varphi = \textit{fresh} \tag{A.54}$$

$$\langle S_Q, \Delta_Q\rangle = \text{sppc}\,(Q, (S_P\ \Gamma)\cup\{y:\varphi\}) \tag{A.55}$$

$$A = \textit{freshInstance typeof}\ \alpha\ (S_Q\ \Delta_P) \tag{A.56}$$

$$B = (S_Q\ \varphi) \tag{A.57}$$

$$C = \textit{freshInstance typeof}\ x\ (S_Q{\circ}S_P\ \Gamma) \tag{A.58}$$

$$S_u = \textit{unify}\ C\ A{\rightarrow}B \tag{A.59}$$

$$\langle S_c, \Delta_c\rangle = \textit{unifyGenContexts}\ (S_u{\circ}S_Q\ \Delta_P\backslash\alpha)\ (S_u\ \Delta_Q) \tag{A.60}$$

$$S_r = (S_c{\circ}S_u{\circ}S_Q{\circ}S_P \cap \textit{atoms}(\Gamma)) \tag{A.61}$$

$$\text{sppc}\,(P\widehat{\alpha}\,[y]\,\widehat{x}Q, \Gamma) = \langle S_r, \Delta_c\rangle \tag{A.62}$$

By induction, twice (using $(Eq.\ A.53)$ *and* $(Eq.\ A.55)$ *with* $(Eq.\ A.57)$*), we obtain:*

$$P :\cdot (S_P\ \Gamma) \vdash_{\text{SP}} \Delta_P \tag{A.63}$$

$$Q :\cdot (S_q{\circ}S_p\ \Gamma), y:B \vdash_{\text{SP}} \Delta_Q \tag{A.64}$$

By Proposition 4.3.7 (1) and $(Eq.\ A.63)$*, we obtain*

$$P :\cdot (S_Q{\circ}S_P\ \Gamma) \vdash_{\text{SP}} (S_Q\ \Delta_P) \tag{A.65}$$

Using Proposition 4.3.7 (5) with $(Eq.\ A.56)$ *(applying Proposition 4.3.7 (2) if necessary), we obtain*

$$P :\cdot (S_Q{\circ}S_P\ \Gamma) \vdash_{\text{SP}} (S_Q\ \Delta_P\backslash\alpha), \alpha : A \tag{A.66}$$

Now, let $\overline{C}$ $=$ *typeof* x *typeof* x $(S_Q{\circ}S_P\ \Gamma)$ *(and so* C $=$ *freshInst*$(\overline{C})$*, by* $(Eq.\ A.58)$*). By definition of freshInst, we have* $\overline{C}{\succeq}C$*. By Proposition 4.3.5 (4) and using* $(Eq.\ A.59)$*, we have*

$$(S_c{\circ}S_u\ \overline{C}){\succeq}(S_c{\circ}S_u\ C) = (S_c{\circ}S_u\ A{\rightarrow}B) = ((S_c{\circ}S_u\ A){\rightarrow}S_c{\circ}S_u\ B) \tag{A.67}$$

By applying Proposition 4.3.7 (1) twice, to $(Eq.\ A.66)$ *and* $(Eq.\ A.64)$*, we*

obtain:

$$P :\cdot (S_c \circ S_u \circ S_Q \circ S_P\ \Gamma) \vdash_{\text{SP}} (S_c \circ S_u \circ S_Q\ \Delta_P \backslash \alpha), \alpha : (S_c \circ S_u\ A) \quad \text{(A.68)}$$

$$Q :\cdot (S_c \circ S_u \circ S_Q \circ S_P\ \Gamma), y : (S_c \circ S_u\ B) \vdash_{\text{SP}} (S_c \circ S_u\ \Delta_Q) \quad \text{(A.69)}$$

By the soundness of unifyGenContexts (Proposition 4.3.22 (1)), we have that $(S_c \circ S_u \circ S_Q\ \Delta_P \backslash \alpha) \succeq \Delta_C$ *and* $(S_c \circ S_u\ \Delta_Q) \succeq \Delta_C$*. Therefore, by Proposition 4.3.7 (6), we obtain that both* $P :\cdot (S_c \circ S_u \circ S_Q \circ S_P\ \Gamma) \vdash_{\text{SP}} \Delta_C, \alpha : (S_c \circ S_u\ A)$ *and* $Q :\cdot (S_c \circ S_u \circ S_Q \circ S_P\ \Gamma), y : (S_c \circ S_u\ B) \vdash_{\text{SP}} \Delta_C$*. Using* $(Eq.\ A.67)$ *and the rule* $(\rightarrow\mathcal{L})$*, we obtain* $P\widehat{\alpha}\,[x]\,\widehat{y}Q :\cdot (S_c \circ S_u \circ S_Q \circ S_P\ \Gamma) \vdash_{\text{SP}} \Delta_C$*, and we conclude by Lemma 4.3.19 (1).*

$R = \widehat{x}P \cdot \beta$**:** *Similar to the* $\widehat{x}P\widehat{\alpha} \cdot \beta$ *case.*

$R = x \cdot P\widehat{\alpha}$**:** *Similar to the* $P\widehat{\alpha}\,[x]\,\widehat{y}Q$ *case.*

$R = P\widehat{\alpha} \dagger \widehat{x}Q$**:** *By induction, twice, we obtain that both* $P :\cdot (S_P\ \Gamma) \vdash_{\text{SP}} \Delta_P$ *and* $Q :\cdot (S_Q \circ S_P\ \Gamma), x : (S_Q\ \overline{A}) \vdash_{\text{SP}} \Delta_Q$*. By weakening (Proposition 4.3.7 (2)) as necessary, we obtain* $P :\cdot (S_P\ \Gamma) \vdash_{\text{SP}} (\Delta_P \backslash \alpha), \alpha : \overline{A}$*. Then, by applying Proposition 4.3.7 (1), twice,* $P :\cdot (S_c \circ S_Q \circ S_P\ \Gamma) \vdash_{\text{SP}} (S_c \circ S_Q\ \Delta_P \backslash \alpha), \alpha : (S_c \circ S_Q\ \overline{A})$ *and* $Q :\cdot (S_c \circ S_Q \circ S_P\ \Gamma), x : (S_c \circ S_Q\ \overline{A}) \vdash_{\text{SP}} (S_c\ \Delta_Q)$*.*

By the soundness of unifyGenContexts (Proposition 4.3.22 (1)), and Proposition 4.3.7 (6), we obtain that both $P :\cdot (S_c \circ S_Q \circ S_P\ \Gamma) \vdash_{\text{SP}} \Delta_c, \alpha : (S_c \circ S_u\ \overline{A})$ *and* $Q :\cdot (S_c \circ S_Q \circ S_P\ \Gamma), x : (S_c \circ S_Q\ \overline{A}) \vdash_{\text{SP}} \Delta_c$*. By applying the rule* (cut)*, we obtain* $P\widehat{\alpha} \dagger \widehat{x}Q :\cdot (S_c \circ S_Q \circ S_P\ \Gamma) \vdash_{\text{SP}} \Delta_c$*. We conclude by applying Lemma 4.3.19 (1), since* $S_r = (S_c \circ S_Q \circ S_P \cap \mathit{atoms}(\Gamma))$*.*

2. *By induction on the structure of the term* R*.*

$R = \langle x.\alpha \rangle$**:** *By Lemma 4.3.8 (1), we must have* $\Gamma = \Gamma', x : \overline{A}$ *and* $\Delta = \alpha : \overline{B}, \Delta'$ *with* $(S\ \overline{A}) \succeq \overline{B}$*. Since* $\Delta_R = \{\alpha : (S\ \overline{A})\}$*, and* $S_R = id$*, can choose* $S' = S$ *and then we have we have* $(S'\ \Delta_R) \succeq \Delta$ *as required.*

$R = \widehat{x}P\widehat{\alpha}\cdot\beta$**:** *From the definition of the algorithm, we have:*

$$\text{sppc}\,(\widehat{x}P\widehat{\alpha}\cdot\beta, \Gamma) = \langle S_r, (S_u\ \Delta_P \backslash \alpha \backslash \beta) \cup \{\beta : \overline{D}\}\rangle \tag{A.70}$$

$$\varphi = \textit{fresh} \tag{A.71}$$

$$\langle S_P, \Delta_P\rangle = \text{sppc}\,(P, \Gamma \cup \{x : \varphi\}) \tag{A.72}$$

$$A = (S_P\ \varphi) \tag{A.73}$$

$$B = \textit{freshInstance typeof}\ \alpha\ \Delta_P \tag{A.74}$$

$$\overline{C} = \forall\textit{-closure}\ A{\rightarrow}B\ \langle (S_P\ \Gamma); \Delta_P \backslash \alpha\rangle \tag{A.75}$$

$$\langle S_u, \overline{D}\rangle = \begin{cases} \textit{unifyGen}\ \overline{C}\ \textit{typeof}\ \beta\ \Delta_P & \textit{if}\ \beta \in \Delta_P \\ \langle \textit{id}, \overline{C}\rangle & \textit{otherwise} \end{cases} \tag{A.76}$$

$$S_r = (S_u \circ S_P \cap \textit{atoms}(\Gamma)) \tag{A.77}$$

By Lemma 4.3.8 (2), we must have $\Delta = \beta : \overline{G}, \Delta'$ *and there exist* E,F *such that*

$$P :\cdot (S\ \Gamma), x : E \vdash_{\text{SP}} \alpha : F, \Delta' \tag{A.78}$$

$$E{\rightarrow}F \lhd_{\langle (S\ \Gamma); \Delta'\rangle} \overline{G} \tag{A.79}$$

Define $S_E = \{(\varphi \mapsto E)\}$*. By construction,* $(S_E \circ S\ \Gamma, x : \varphi) = ((S\ \Gamma), x : E)$*. By induction, using* $(Eq.\ A.72)$*, there exists* S_1 *such that*

$$S_E \circ S = S_1 \circ S_P \tag{A.80}$$

$$(S_1\ \Delta_P) \succeq (\Delta', \alpha : F) \tag{A.81}$$

Let $\overline{B} =$ *typeof* $\alpha\ \Delta_P$ *(so that, by* $(Eq.\ A.74)$*,* $B =$ *freshInst*$(\overline{B})$*). By Proposition 4.3.5 (7), there exists* S_2 *such that dom*(S_2) *consists of only the fresh atomic types in* $B =$ *freshInst*$(\overline{B})$*, and* $(S_2 \circ S_1\ B) = F$*. Now we have*

$$(S_2 \circ S_1\ A{\rightarrow}B) = E{\rightarrow}F \tag{A.82}$$

$$(S_2 \circ S_1\ \Delta_P \backslash \alpha) = (S_1\ \Delta_P \backslash \alpha) \succeq \Delta' \tag{A.83}$$

By using $(Eq.\ A.75)$ *with Proposition 4.3.16 (3), we are able to show that* $(S_2 \circ S_1\ \overline{C}) \succeq \forall$*-closure* $(S_2 \circ S_1\ A{\rightarrow}B)\ (S_2 \circ S_1\ \langle (S_P\ \Gamma); \Delta_P \backslash \alpha\rangle)$*, and, by our knowledge of dom*(S_2) *and using* $(Eq.\ A.82)$*, we can simplify this to obtain* $(S_1\ \overline{C}) \succeq \forall$*-closure* $E{\rightarrow}F\ \langle (S_1 \circ S_P\ \Gamma); (S_1\ (\Delta_P \backslash \alpha))\rangle$*. Now, by* $(Eq.\ A.81)$*, we have* $(S_1\ (\Delta_P \backslash \alpha)) \succeq \Delta'$*. Using this fact, along with* $(Eq.\ A.80)$ *and the fact that dom*$(S_E) = \{\varphi\}$*, we can apply Proposition 4.3.16 (4) to obtain that* $(S_1\ \overline{C}) \succeq \forall$*-closure* $E{\rightarrow}F\ \langle (S\ \Gamma); \Delta'\rangle$*. By Proposition 4.3.16 (1), using*

$(Eq.\ A.79)$, we obtain $\forall$-closure $E{\rightarrow}F\ \langle(S\,\Gamma);\Delta'\rangle{\succeq}\overline{G}$, and so by Proposition 4.3.5 (1) we have

$$(S_1\ \overline{C}){\succeq}\overline{G} \tag{A.84}$$

We claim that we can now show that, for some substitution S_3 satisfying $S_3{\circ}S_u = S_1$, we have $(S_3\ \overline{D}){\succeq}\overline{G}$ and $(S_3\ (S_u\ (\Delta_P{\setminus}\alpha))){\succeq}\Delta'$, from which (as we shall then show) we can complete the case easily. We consider two cases:

- *$(\beta \in \Delta_P)$: Then, by $(Eq.\ A.81)$, we have $\beta \in \Delta'$. Since $\Delta = \beta:\overline{G}, \Delta'$, we must have $\beta:\overline{G} \in \Delta'$. Now, let $\overline{H}$ = typeof $\beta\ \Delta_P$. Then $(S_1\ \overline{H}){\succeq}\overline{G}$. By $(Eq.\ A.84)$ and Theorem 4.3.21 (2) (and following $(Eq.\ A.76)$), there exists S_3 such that $S_3{\circ}S_u = S_1$ and $(S_3\ \overline{D}){\succeq}\overline{G}$, and so, by $(Eq.\ A.81)$, we obtain $(S_3{\circ}S_u\ (\Delta_P{\setminus}\alpha)){\succeq}\Delta'$ as claimed.*
- *$(\beta \notin \Delta_P)$: Then, by $(Eq.\ A.76)$, $(S_u, \overline{D}) = (id, \overline{C})$. Let $S_3 = S_1$, and then trivially we have $S_3{\circ}S_u = S_1$ and $(S_3\ \overline{D}){\succeq}\overline{G}$ (from $(Eq.\ A.84)$) and $(S_3{\circ}S_u\ (\Delta_P{\setminus}\alpha)){\succeq}\Delta'$ (by $(Eq.\ A.81)$).*

Therefore, in both cases, we have:

$$(S_3\ \overline{D}){\succeq}\overline{G} \tag{A.85}$$

$$(S_3\ (S_u\ (\Delta_P{\setminus}\alpha))){\succeq}\Delta' \tag{A.86}$$

$$S_3{\circ}S_u = S_1 \tag{A.87}$$

Therefore, we can deduce $(S_3\ (S_u\ (\Delta_P{\setminus}\alpha{\setminus}\beta))){\succeq}(\Delta'{\setminus}\beta)$, and so it follows that $(S_3\ (S_u\ (\Delta_P{\setminus}\alpha{\setminus}\beta)), \beta:\overline{D}){\succeq}\Delta'$ as needed. Finally, by combining $(Eq.\ A.80)$ with $(Eq.\ A.87)$, and applying Lemma 4.3.19 (7), we obtain $S_E{\circ}S = S_1{\circ}S_P = S_3{\circ}S_u{\circ}S_P = (S_3{\circ}((S_u{\circ}S_P){\setminus}$atoms$(\Gamma))){\circ}((S_u{\circ}S_P) \cap$ atoms$(\Gamma))$. Now, noting that dom$(S_E) = \{\varphi\}$, we apply Lemma 4.3.18 (2) and deduce that there exists a substitution S_5 such that $S = S_5{\circ}((S_u{\circ}S_P) \cap$ atoms$(\Gamma))$, as required.

$R = P\widehat{\alpha}\,[x]\,\widehat{y}Q$**:** *In accordance with the algorithm, we have:*

$$\langle S_P, \Delta_P\rangle = \text{sppc}\,(P, \Gamma) \tag{A.88}$$
$$\varphi = \textit{fresh} \tag{A.89}$$
$$\langle S_Q, \Delta_Q\rangle = \text{sppc}\,(Q, (S_P\ \Gamma)\cup\{y:\varphi\}) \tag{A.90}$$
$$A = \textit{freshInstance typeof}\ \alpha\ (S_Q\ \Delta_P) \tag{A.91}$$
$$B = (S_Q\ \varphi) \tag{A.92}$$
$$C = \textit{freshInstance typeof}\ x\ (S_Q{\circ}S_P\ \Gamma) \tag{A.93}$$
$$S_u = \textit{unify}\ C\ A{\rightarrow}B \tag{A.94}$$
$$\langle S_c, \Delta_c\rangle = \textit{unifyGenContexts}\ (S_u{\circ}S_Q\ \Delta_P{\setminus}\alpha)\ (S_u\ \Delta_Q) \tag{A.95}$$
$$S_r = (S_c{\circ}S_u{\circ}S_Q{\circ}S_P \cap \textit{atoms}(\Gamma)) \tag{A.96}$$
$$\langle S_r, \Delta_c\rangle = \text{sppc}\,(P\widehat{\alpha}\,[y]\,\widehat{x}Q, \Gamma) \tag{A.97}$$

By Lemma 4.3.8 (3), we have, for some Γ',$\overline{D}$,E *and* F*, that*

$$\Gamma = \Gamma', x:\overline{D} \tag{A.98}$$
$$(S\ \overline{D}){\succeq}(E{\rightarrow}F) \tag{A.99}$$
$$P :\cdot\ (S\ \Gamma') \vdash_{\text{SP}} \alpha : E, \Delta \tag{A.100}$$
$$Q :\cdot\ (S\ \Gamma'), y:F \vdash_{\text{SP}} \Delta \tag{A.101}$$

For reference, we explicitly write

$$\overline{D} = \overrightarrow{\forall X_i.}\,D \tag{A.102}$$

By induction, using $(Eq.\ A.88)$ *and* $(Eq.\ A.100)$*, there exists* S_1 *such that:*

$$S = S_1{\circ}S_P \tag{A.103}$$
$$(S_1\ \Delta_P){\succeq}(\alpha:E, \Delta) \tag{A.104}$$

By $(Eq.\ A.98)$*,* $(Eq.\ A.101)$ *and weakening (Proposition 4.3.7 (2)) as necessary, we obtain*

$$Q :\cdot\ (S\ \Gamma), y:F \vdash_{\text{SP}} \Delta \tag{A.105}$$

Now, let

$$S_F = \{(\varphi \mapsto F)\} \tag{A.106}$$

Then $(S_F{\circ}S_1\ ((S_P\ \Gamma), y:\varphi)) = ((S\ \Gamma), y:F)$ *by construction. By* $(Eq.\ A.90)$

and $(Eq.\ A.105)$*, and by induction, there exists* S_2 *such that:*

$$S_F \circ S_1 = S_2 \circ S_Q \tag{A.107}$$

$$(S_2\ \Delta_Q) \succeq \Delta \tag{A.108}$$

Now define (respecting $(Eq.\ A.91)$*):*

$$\overline{A} = \textit{typeof}\ \alpha\ (S_Q\ \Delta_P) \tag{A.109}$$

$$A = \textit{freshInst}(\overline{A}) \tag{A.110}$$

By $(Eq.\ A.102)$ *and* $(Eq.\ A.93)$*,* $C = (S_Q \circ S_P\ D[\overrightarrow{\varphi_i/X_i}])$ *for fresh* $\overrightarrow{\varphi_i}$*. Now, using* $(Eq.\ A.107)$ *and* $(Eq.\ A.103)$*, we have*

$$S_2 \circ S_Q \circ S_P = S_F \circ S_1 \circ S_P = S_F \circ S \tag{A.111}$$

and so $(S_2\ C) = (S_2\ S_Q \circ S_P\ D[\overrightarrow{\varphi_i/X_i}]) = (S\ D)[\overrightarrow{\varphi_i/X_i}]$ *given* $(Eq.\ A.106)$*. By* $(Eq.\ A.99)$ *and Proposition 4.3.5 (7), there exists* S_3 *such that*

$$\textit{dom}(S_3) = \{\overrightarrow{\varphi_i}\} \tag{A.112}$$

$$(S_3 \circ S_2\ C) = (E \rightarrow F) \tag{A.113}$$

By $(Eq.\ A.104)$*,*$(Eq.\ A.89)$ *and* $(Eq.\ A.108)$ *we get* $(S_F \circ S_1\ \Delta_P) \succeq (\alpha : E, \Delta)$*. By* $(Eq.\ A.107)$*, this means that* $(S_2 \circ S_Q\ \Delta_P) \succeq (\alpha : E, \Delta)$*, and in particular, by* $(Eq.\ A.109)$*, we have* $(S_2\ \overline{A}) \succeq E$*. By Proposition 4.3.5 (7) and* $(Eq.\ A.110)$ *(in which, say* $\{\overrightarrow{\varphi_j}\}$ *are the fresh atomic types chosen), there exists* S_4 *such that*

$$\textit{dom}(S_4) = \{\overrightarrow{\varphi_j}\} \tag{A.114}$$

$$(S_4 \circ S_2\ A) = E \tag{A.115}$$

(note that $(Eq.\ A.110)$ *implies that* $(S_2\ A) = \textit{freshInst}(S_2\ \overline{A})$ *up to choice of fresh atomic types, given that* S_2 *does not clash with the atomic types chosen).*

By $(Eq.\ A.112)$*,* $(Eq.\ A.114)$ *and* $(Eq.\ A.115)$*, we deduce* $(S_4 \circ S_3 \circ S_2\ A) = E$*. Also, by* $(Eq.\ A.92)$*,* $(S_4 \circ S_3 \circ S_2\ B) = (S_4 \circ S_3 \circ S_2 \circ S_Q\ \varphi) = F$*. By* $(Eq.\ A.113)$*, we have* $(S_4 \circ S_3 \circ S_2\ C) = E \rightarrow F = (S_4 \circ S_3 \circ S_2\ A \rightarrow B)$*. By completeness of unification (Lemma 3.3.9 (2)), there exists a substitution* S_5 *such that*

$$S_4 \circ S_3 \circ S_2 = S_5 \circ S_u \tag{A.116}$$

Now, using $(Eq.\ A.116)$*,* $(Eq.\ A.107)$*,* $(Eq.\ A.112)$ *and* $(Eq.\ A.114)$*, we*

obtain:

$$\begin{array}{ll}
(S_5 \circ S_u \circ S_Q\ \Delta_P \backslash \alpha) & \\
= (S_4 \circ S_3 \circ S_2 \circ S_Q\ \Delta_P \backslash \alpha) & (Eq.\ A.116) \\
= (S_4 \circ S_3 \circ S_F \circ S_1\ \Delta_P \backslash \alpha) & (Eq.\ A.107) \\
= (S_1\ \Delta_P \backslash \alpha) & (Eq.\ A.112), (Eq.\ A.114), (Eq.\ A.106) \\
\succeq \Delta & (Eq.\ A.104)
\end{array}$$

Similarly, we deduce $(S_5 \circ S_u\ \Delta_Q) = (S_2\ \Delta_Q) \succeq \Delta$ *using* $(Eq.\ A.108)$*. Therefore, by* $(Eq.\ A.95)$*, there exists* S_6 *with* $S_5 = S_6 \circ S_5$ *and* $(S_6\ \Delta_C) \succeq \Delta$*. Now,* $S_4 \circ S_3 \circ S_F \circ S = S_4 \circ S_3 \circ S_2 \circ S_Q \circ S_P = S_5 \circ S_u \circ S_Q \circ S_P = S_6 \circ S_c \circ S_u \circ S_Q \circ S_P$*. Therefore,* $S_4 \circ S_3 \circ S_F \circ S = S_6 \circ ((S_c \circ S_u \circ S_Q \circ S_P) \backslash \textit{atoms}(\Gamma)) \circ ((S_c \circ S_u \circ S_Q \circ S_P) \cap \textit{atoms}(\Gamma))$ *by Lemma 4.3.19 (7). By applying Lemma 4.3.18 (2), using* $(Eq.\ A.112)$*,* $(Eq.\ A.114)$ *and* $(Eq.\ A.106)$*, there exists* S_7 *such that* $S = S_7 \circ S_r$ *as required.*

$R = \widehat{x}P \cdot \beta$**:** *Similar to the* $\widehat{x}P\widehat{\alpha} \cdot \beta$ *case.*

$R = x \cdot P\widehat{\alpha}$**:** *Similar to the* $P\widehat{\alpha}\,[x]\,\widehat{y}Q$ *case.*

$R = P\widehat{\alpha} \dagger \widehat{x}Q$**:** *In accordance with the algorithm, we have:*

$$\langle S_P, \Delta_P \rangle = \text{sppc}\,(P, \Gamma) \tag{A.117}$$
$$\overline{A} = \textit{typeof}\ \alpha\ \Delta_P \tag{A.118}$$
$$\langle S_Q, \Delta_Q \rangle = \text{sppc}\,(Q, (S_P\ \Gamma) \cup \{x : \overline{A}\}) \tag{A.119}$$
$$\langle S_c, \Delta_c \rangle = \textit{unifyGenContexts}\ (S_Q\ \Delta_P \backslash \alpha)\ \Delta_Q \tag{A.120}$$
$$S_r = (S_c \circ S_Q \circ S_P \cap \textit{atoms}(\Gamma)) \tag{A.121}$$
$$\text{sppc}\,(P\widehat{\alpha} \dagger \widehat{x}Q, \Gamma) = \langle S_r, \Delta_c \rangle \tag{A.122}$$

By Lemma 4.3.8 (6), there exists $\overline{B}$ *such that*

$$P :\cdot (S\ \Gamma) \vdash_{\text{SP}} \alpha : \overline{B}, \Delta \tag{A.123}$$
$$Q :\cdot (S\ \Gamma), x : \overline{B} \vdash_{\text{SP}} \Delta \tag{A.124}$$

By induction, using $(Eq.\ A.117)$*, there exists* S_1

$$S = S_1 \circ S_P \tag{A.125}$$
$$(S_1\ \Delta_P) \succeq (\alpha : \overline{B}, \Delta) \tag{A.126}$$

By $(Eq.\ A.118)$*,* $(S_1\ \overline{A}) \succeq \overline{B}$*. By* $(Eq.\ A.124)$ *and Proposition 4.3.7 (4), we*

obtain

$$Q :\cdot (S\ \Gamma), x : (S_1\ \overline{A}) \vdash_{\text{SP}} \Delta \tag{A.127}$$

Note that

$$(S_1\ ((S_P\ \Gamma), x : \overline{A})) = ((S\ \Gamma), x : (S_1\ \overline{A})) \tag{A.128}$$

Therefore, by induction, using $(Eq.\ A.119)$*, there exists* S_2 *such that*

$$S_1 = S_2 \circ S_Q \tag{A.129}$$

$$(S_2\ \Delta_Q) \succeq \Delta \tag{A.130}$$

By $(Eq.\ A.126)$*, we have* $(S_2\ S_Q\ \Delta_P \backslash \alpha) \succeq \Delta$*. Using* $(Eq.\ A.126)$*,* $(Eq.\ A.130)$*,* $(Eq.\ A.120)$ *and Theorem 4.3.21 (2), there exists* S_3 *with* $S_2 = S_3 \circ S_c$*. Using* $(Eq.\ A.125)$ *and* $(Eq.\ A.129)$*, we obtain as required:*

$$\begin{aligned} S &= S_3 \circ S_c \circ S_Q \circ S_P \\ &= (S_3 \circ ((S_c \circ S_Q \circ S_P) \backslash \mathit{atoms}(\Gamma))) \circ ((S_c \circ S_Q \circ S_P) \cap \mathit{atoms}(\Gamma)) \end{aligned}$$

□